ELSEVIER

Science of Computer Programming 24 (1995) 249-286

Science of Computer Programming

A mathematical definition of full Prolog

Egon Börger [a,*], Dean Rosenzweig [b]

[a] *Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56100 Pisa, Italy*

[b] *University of Zagreb, FSB, Salajeva 5, 41000 Zagreb, Croatia*

Received November 1992; revised May 1993

Communicated by K.R. Apt

Abstract

The paper provides a mathematical yet simple model for the full programming language Prolog, as apparently intended by the ISO draft standard proposal. The model includes all control constructs, database operations, solution collecting predicates and error handling facilities, typically ignored by previous theoretical treatments of the language. We add to this the ubiquitous box-model debugger. The model directly reflects the basic intuitions underlying the language and can be used as a primary mathematical definition of Prolog. The core of the model has been applied for mathematical analysis of implementations, for clarification of disputable language features and for specifying extensions of the language in various directions. The model may provide guidance for extending the established theory of logic programming to the extralogical features of Prolog.

1. Introduction

One of the original aims of mathematical semantics was to provide the programmer with a set of mathematical models and tools, helping him to express his intuitions, and reason about them, in a precise and secure way — very much like any other engineer. The pioneers of "mathematical semantics for computer languages" had explicitly meant *real languages* and their implementations.

> An essential topic will be the discussion of the relation between the mathematical semantics for a language and the implementation of the language. What we claim the mathematics will have provided is the standard against which to judge an implementation. [44, p. 40]

* Corresponding author. Email: boerger@di.unipi.it.

SSDI 0167-6423(95)00006-2

This paper provides a mathematical model for a real language, Prolog, as apparently intended by the draft standard proposal [47]. The model directly reflects the basic intuitions underlying the language, providing them with a mathematically rigorous yet simple formulation.

Since we are

> ... concerned with bringing real theory to apply to real programming ...
> [6, p. III]

the model covers the full language, i.e. not only the Horn clause logic core, but all control constructs (*true, fail, cut, call, metacall, once, and, or, if_then, if_then_else, not, repeat, catch, throw*), all database operations (*asserta, assertz, clause, retract, current_predicate, abolish*), all solution collecting predicates (*findall, bagof, setof*) and error handling features of the draft standard proposal, typically ignored by previous theoretical treatments of the language. On top of this we add the ubiquitous box-model debugger. The Prolog features we skip (syntax, operating system interface, arithmetics) are not in any way problematic for our methodology — they are skipped since they are not characteristic either of logic programming or of Prolog, and can be dealt with in a straightforward manner [9].

The core of the model has been successfully applied already for mathematical analysis of implementations, for clarification of disputable language features and for specifying extensions of the language in various directions.

In fact, it is from the core of the present model that a mathematical reconstruction of a generally accepted implementation method for Prolog, the Warren Abstract Machine [45,1], was formally derived, and proved to execute Prolog correctly — with respect to the model — given explicit mathematical assumptions about the compiler [17].

The core of the present specification was mapped to, and served as foundation for, specifying extensions of Prolog and their implementation, such as

- Protos-L — Prolog enriched with polymorphic types [5], where even the correctness proof for the implementation could be uniformly extended from the WAM;
- CLPR — constraint logic programming system [19], where also the correctness proof for the implementation could be uniformly extended from the WAM;
- Prolog III — Prolog with constraints [20];
- Babel — Prolog enriched with functional expressions [10];
- Parlog, Concurrent Prolog — concurrent logic programming languages, [13,14];
- object-oriented extension of Prolog [36].

It has also provided a framework for identifying and clarifying disputable language features and related implementation issues, such as the problem of semantics of dynamic database operations [11,16] and solution-collecting predicates [18].

We view the model as a primary direct formalization of the basic intuitions, concepts and operations of the language, as understood by its practitioners and implementors. This is not to say that we would accept any particular implementation as being a definition of the language. It is the other way round:

> ... unless there is a prior, generally accepted mathematical definition of a language at hand, who is to say whether a proposed implementation is *correct*? [43, p. 2]

Development of a primary model releases us from any obligation of proof — it however places us under a (much more severe) obligation to abstract into mathematical form the central common ideas underlying current implementations and verbal descriptions. Here the challenge is that of *adequacy*, rather than correctness. Thus the model has to be transparent; the central common ideas should be recognizable by inspection, so to speak.

The kind of transparence needed can be achieved only if the methodology allows modelling on precisely the abstraction level of the language.

> ... the specification of requirements should be formulated from the beginning at the highest possible level of abstraction, using all the available power of mathematics ... [29, p.VII]

This implies that basic concepts have to be expressed directly, without encoding, taking the objects of the language as abstract entities, such as they appear there. The methodology should thus make abstract data types freely available, i.e. abstract domains of objects with operations. The signature of abstract data types should contain no less, but also no more, than what is present or implicit in the language.

We shall also not attempt to reduce the intuition of actions-in-time to something else, but, on the contrary, try to model it as faithfully as possible. All actions, which (in programming languages) appear as basic, on the given level of abstraction, are local and come with clear preconditions and effects. Their natural formalization should then be based on local modifications, guarded by simple conditions.

These two ideas are captured by the concept of *evolving algebra*, put forward by [27,28]. An evolving algebra is essentially a transition system with statics given by a (first-order) signature, and dynamics given by transition rules which transform structures. Statically it is algebraic, and dynamically it is operational, so that modelling with evolving algebras can be termed "algebraic operational semantics". At each level of abstraction we are free to choose the signature so as to fit tightly the concepts and the intuitions we intend to model, and the transition rules can be chosen so to reflect explicitly the actions identified as basic. In particular, the local character of basic actions can be reflected, without need to refer explicitly to any notion of "global state". Methodological overhead is thus almost nil: there is no encoding, no forcing of objects into a fixed abstraction level, no forcing of dynamics into a fixed static representation. As a consequence, upon application to real (non-toy) systems, the too often experienced combinatorial explosion of formalism and/or mathematics just does not happen.

While it is well known from algebraic specification that the set of operations chosen should match the level of data abstraction, evolving algebras offer an additional degree of freedom, to choose the abstraction level of basic actions. This determines which operations will be considered as static (and represented in the signature) and which as dynamic (by being decomposed into basic steps through transition rules). Roughly,

any action which is "finer grained" than what has been chosen to be a basic step, is static — "coarser grained" actions are dynamic. For instance, on the abstraction level of the model of this paper, unification is a static function, available inside a basic step, while on WAM level [17] unification is a dynamically represented algorithm, involving many finer grained basic steps. In the opposite direction, the fixed-point approach to dynamics could, from this perspective, be viewed as the limit case of an infinite basic step. In Plotkin's structural operational semantics [40] proof rules for basic actions directly reflect the syntactic structure of the program; thus SOS looses some of its nice "subformula properties" as soon as the language imposes a notion of basic action which is not so tightly coupled to the syntax.

The freedom of choosing *action abstraction*, provided by evolving algebras, allows us to match it naturally to data abstraction. To pursue the unification example, static unification comes naturally together with abstract (unencoded) notions of term and substitutions (representing themselves, so to say), whereas dynamic analysis of unification, as provided by WAM, comes naturally with representation of terms and substitutions as complex composite objects, encoded on the heap.

(Evolving) algebras are deeply rooted in traditional mathematics. Nevertheless, evolving algebra descriptions can be readily understood as "pseudocode over abstract data", and followed without any formal training in logic, as we have experienced with many programmers and implementors.

The question might be raised, why, in a market full of formal systems, use yet another one? To summarize, no other formalism we know of provides the simplicity, freedom of both data and action abstraction and a capability of faithful modelling of involved dynamics, as offered by evolving algebras. No methodology will, by itself, make the difficulties of analyzing complex programs disappear; evolving algebras at least do not introduce extraneous difficulties.

Since evolving algebras are just algebras, in the usual sense of mathematics, we may use any mathematical techniques whatsoever to prove their properties, and hence also the properties of computational phenomena they reflect (in this case Prolog). We do not, at this point, propose any fixed proof methodology — the role of proof systems is to single out characteristic proof patterns (for study and/or use). Such characteristic patterns, in the case of evolving algebras, have yet to be identified. See however [17], where a proof pattern seems to emerge, from an analysis of a class of complex programs, i.e. Prolog-to-WAM compilers. See also [12], where more explicit proof-patterns result from the general theory of communicating evolving algebras developed in [26].

The preceding discourse on methodology is motivated by the ambition to provide a *transparent primary mathematical model for the full language*. There would be no need for a new model, were we interested only in a slight extension of Horn clause programmming, (by say *cut, and, or*), since many models of such fragments exist in the literature, realized by different methodologies, such as [2,21–23,30,31,35,37,38,42]. It is not at all clear whether or how any of these models could be extended to cover the full language. Attempts to cover the full language are far less numerous [4,25,39]. These models are however neither primary nor transparent — they consist in technically

involved reductions to some other formalism, which can hardly be understood as adequate expression of basic intuitions. The same remark applies to modelling by code written in another programming language, which also just shifts the semantical burden.

Our model, when restricted to pure Prolog, is easily linked to the established body of logic programming theory, cf. Prolog Tree Theorem below. Our approach may in fact provide guidance for extending the theory of Horn-clause logic to the real programming language. By providing a mathematical model for the latter, we hope to help reduce the

> ... mismatch between theory and practice ... that much of the theory of logic programming only applies to pure subsets of Prolog, whereas the extra-logical facilities of the language appear essential for it to be practical. [34]

This paper synthesizes, streamlines and considerably extends the work reported in preliminary papers [7–9,15].

The paper is organized as follows. Section 2 introduces what is used of the underlying framework of evolving algebras. Section 3 develops the core part of the model, dealing with SLD-resolution fragment of Prolog. The model is based on the widespread intuitive picture of Prolog trees, which is close to its proof theoretical background. Unlike SLD-tree, however, the Prolog trees are finite and grow incrementally, governed by four rules. Section 4 defines all control constructs of the draft standard proposal; the tree model allows simple dynamics, given by one rule per construct. Section 5 defines all the constructs for inspection and modification of the dynamic Prolog database (program) of the draft standard proposal. Section 6 defines the solution collecting predicates, and Section 7 the error handling facilities. The techniques developed for these sections give us a mathematical definition of the box-model debugger for free, as given in Section 8. An appendix lists for reference the full signature (except for syntactic functions, which are taken for granted) and all rules defining the model.

2. Evolving algebras

The Prolog model constructed in this paper is an *evolving algebra*, which is a notion introduced by [28]. Since this notion is a mathematically rigorous form of fundamental operational intuitions of computing, the paper can be followed without any particular theoretical prerequisites. The reader who is not interested in foundational issues, might read our rules as "pseudocode over abstract data". However, remarkably little is needed for full rigour — the definitions listed in this section suffice.

The abstract data come as elements of sets (domains, *universes*). The operations allowed on universes will be represented by partial *functions*.

We shall allow the setup to *evolve* in time, by executing *function updates* of form

$$f(t_1, \ldots, t_n) := t$$

whose execution is to be understood as *changing* (or defining, if there was none) the value of function f at given arguments. The 0-ary functions will then be something like

variable quantities of ancient mathematics or *variables* of programming, which explains why we are reluctant to call them *constants*.

The precise way our "abstract machines" (evolving algebras) may evolve in time will be determined by a finite set of *transition rules* of form

if $R?$ **then** $R!$

where $R?$ (condition or guard) is a boolean, the truth of which triggers *simultaneous* execution of all updates in the finite set of updates $R!$. Simultaneous execution helps us avoid fussing and coding to, say, interchange two values. Since functions may be partial, equality in the guards is to be interpreted in the sense of "partial algebras" [46], as implying that both arguments are defined (see also [26]).

More precisely,

Definition. An *evolving algebra* is given by a finite set of transition rules.

The signature of a rule, or that of an evolving algebra, can always be reconstructed, as the set of function symbols occurring there.

Definition. Let A be an evolving algebra. A *static algebra* of A is any algebra of $\Sigma(A)$, i.e. a pair (U, I) where U is a set and I is an interpretation of $\Sigma(A)$ with partial functions over U.

In applications an evolving algebra usually comes together with a set of *integrity constraints*, i.e. extralogical axioms and/or rules of inference, specifying the intended domains. We tacitly understand the notion of interpretation as validating any integrity constraints imposed.

Our rules will always be constructed so that the guards imply *consistency* of updates, cf. [26] for discussion. While the effect of executing a rule in a static algebra is intuitively clear, it is given precisely by

Definition. The *effect* of updates $R! = \{f_i(\vec{s}_i) := t_i \mid i = 1, \ldots, n\}$, consistent in an algebra (U, I), is to transform it to $(U, I_{R!})$, where

$$I_{R!}(f)(\vec{y}) \equiv \begin{cases} I(t_i) & \text{if } f \doteq f_i,\ \vec{y} = I(\vec{s}_i),\ i = 1, \ldots, n \\ I(f)(\vec{y}) & \text{otherwise} \end{cases}$$

where $\vec{y}$ is any tuple of values in U of f's arity, and $\doteq$ denotes syntactical identity.

The assumption of consistency ensures that $I_{R!}$ is well defined.

We have now laid down precisely the way in which transition rules transform first-order structures. Evolving algebras can then be understood as *transition systems* (directed graphs) whose states (nodes) are first-order structures, and the transition relation (set of arrows) is given by applicable rules.

Definition. $\mathcal{A} \xrightarrow{R} \mathcal{A}_R$ whenever $\mathcal{A} \models R?$ ("R is applicable in $\mathcal{A}$").

The rules are to be thought of as containing only closed, variable-free terms. We shall nevertheless display rules containing variables, but only as an abbreviational device which enhances readability, and is otherwise eliminable. Say,

if $\ldots a = \langle X, Y \rangle \ldots$
then $\ldots X \ldots Y \ldots$

abbreviates

if $\ldots \mathrm{ispair}(a) \ldots$
then $\ldots \mathrm{fst}(a) \ldots \mathrm{snd}(a) \ldots,$

sparing us the need to write explicitly the recognizers and the selectors.

In applications of evolving algebras (including the present one) one usually encounters a *heterogenous* signature with several universes, which may in general grow and shrink in time — update forms are provided to extend a universe:

extend A **by** $t_1, \ldots, t_n$ **with** *updates* **endextend**

where *updates* may (and should) depend on t_i's, setting the values of some functions on *newly created* elements t_i of A.

Gurevich [28] has however shown how to reduce such setups to the above basic model of a homogenous signature (with one universe) and function updates only (see also [26]).

As Prolog is a sequential language, our rules are organized in such a way that at every moment at most one rule is applicable.

The forms obviously reducible to the above basic syntax, which we shall freely use as abbreviations, are **where** and **if then else**. We shall assume that we have the standard mathematical universes of booleans, integers, lists of whatever etc (as well as the standard operations on them) at our disposal without further mention. We use usual notations, in particular Prolog notation for lists.

An evolving algebra, as given above, determines the dynamics of a very large transition system. We are usually (in particular here) only interested in states reachable from some designated *initial states*, which may be, orthogonally, specified in various ways. We can use an informal mathematical description, like in model theory; we can devise special intitializing evolving algebra rules which, starting form a canonical "empty" state, produce the initial states we need; we may use any formal methods, such as those of algebraic specification.

3. Prolog tree

In this section we lay down the signature and the rules for the pure Prolog core of our model, and prove its correctness wrt SLD resolution.

3.1. Signature

A Prolog computation can be seen as systematic search of a space of possible solutions to an initially given query. The set of computation states is often viewed as carrying a tree structure, with the initial state at the root, and *son* relation representing alternative (single) resolution steps. We then represent *Prolog computation states* in a set *NODE* with its two distinguished elements *root* and *currnode*, with the latter representing the (dynamically) current state. Each element of *NODE* has to carry all information relevant — at the desired abstraction level — for the computation state it represents. This information consists in *the sequence of goals* still to be executed, the *substitution* computed so far, and possibly the *sequence of alternative states* still to be tried, as we will explain below.

The tree structure over the universe *NODE* is realized by a function

$$father: NODE - \{root\} \rightarrow NODE$$

such that from each node there is a unique *father* path towards *root*. We do not assume the *tree algebra*

$$(NODE; root, currnode; father)$$

to be platonically given as a static, possibly infinite, object representing the whole search space; we rather create it dynamically as the computation proceeds, out of the initial state (determined by given program and query) as the value of *currnode*, *father*ed by the empty *root*.

When at a given node n the selected literal (activator) *act* is called for execution, for each possible immediate resolvent state a son of n will be created, to control the alternative computation thread. Each son is determined by a corresponding *candidate clause* of the program, i.e. one of those clauses whose head might unify with *act* (given, of course, that the predication is defined by clauses at all, i.e. is *user defined*). All such *candidate sons* are attached to n as a list $cands(n)$, in the order reflecting the ordering of corresponding candidate clauses in the program. We require of course the *cands*-lists to be consistent with *father*, i.e. whenever *Son* is among $cands(Father)$, then $father(Son) = Father$.

This action of augmenting the tree with $cands(n)$ takes place at most once, when n gets first visited. We distinguish this situation using a 0-ary function *mode*, which at that moment has the value *Call*. The mode then turns to *Select*, and a step of resolution is attempted, i.e. the first unifying son from $cands(n)$ gets visited (becomes the value of *currnode*), again in *Call* mode. The selected son is simultaneously deleted from the $cands(n)$ list. If control ever returns to n, (by *backtracking*, cf. below), it will be in *Select* mode, and the next candidate son will be selected, if any.

If none, that is if in *Select* mode $cands(n) = [\ ]$, all attempts at resolution from the state represented by n will have failed, and n will be *abandoned* by returning control to its *father*. This action is usually called *backtracking*. The *father* function then may be seen as representing the structure of Prolog's *backtracking behaviour*.

The *resolution step*, applied to a unifying son from the *cands* list, is executed in terms of the *goal sequence* attached to each node: the body of candidate son's clause is pushed to what is left when the activator is removed - the *continuation*, and the whole goal sequence gets updated by the *mgu*. The goal sequence thus takes the form of a *calling stack*.

What remains for this section is to complete a precise description of the signature (statics) and transition rules (dynamics). We assume the universes of Prolog literals (predications), goals (sequences of literals), terms and clauses

$$LIT, \quad TERM, \quad GOAL = TERM^*, \quad CLAUSE$$

We will use (and consider as part of Prolog tree algebras) all the usual *list operations* for which we adopt standard notation — allowing ourselves the freedom to confuse a list of literals with their iterated conjunction (clause body), suppressing the obvious translation.

Notice that *GOAL* is *not* defined as LIT^* — we allow arbitrary terms just in order to incorporate the *metacall* facility of Prolog, described in Section 4.

The information relevant for determining a computation state will be associated to nodes by appropriate (in general partial) functions on the universe *NODE*.

For each state we have to know the sequence of literals still to be called. Whereas in the definition of the SLD-tree [33,3] this sequence is depicted as flat, it seems closer to procedural intuition to keep it split into clause (procedure) bodies, preserving the structure of "procedure calling stack". In fact, some Prolog constructs — such as *cut, catch, throw, findall* ... — do rely on that structure, by referring explicitly to (some) calling point, and/or by performing special action upon completion of a call. Splitting of goal into a sequence of bodies, on the level of data representation, is coupled, in dynamics, to decomposing of a resolution step to call and select actions, and thus enables uniform handling of user-defined predicates and built-in "extralogical" constructs.

Among the control constructs the *cut* certainly stands out, being — for better or for worse — a characteristic feature of Prolog. We therefore provide a special data structure for execution of *cut*, decorating every goal (clause body) with appropriate *cutpoint*, i.e. backtracking state (node) current when the clause was called. Hence a universe

$$DECGOAL = GOAL \times NODE.$$

For some other constructs, much more special and far less central to Prolog, we rely on the obvious programmer's solution of putting special *marks* on the calling stack just before pushing the procedure (clause) body, indicating the special action to be taken and the calling node it refers to.

The logical integrity of data structures might have been better preserved by avoiding the marks — a mathematically minded reader might prefer to define, by induction, the notion of calling node: the node a mark would point to if it were there. But beware of the *cut*!

Thus a function

$$decglseq : NODE \rightarrow (DECGOAL + MARK)^*$$

associating a sequence of (decorated) goals, interspersed with some marks, to each node. As far as pure Prolog is concerned, both cutpoints and marks can be disregarded, and *decglseq* could simply be a *GOAL*, see Prolog Tree Theorem below.

The substitution current at a state is represented by a function

$$s : NODE \rightarrow SUBST$$

where *SUBST* is a universe of *substitutions*, coming together with the function

$$mgu : TERM \times TERM \rightarrow SUBST \cup \{nil\}$$

where *mgu* is an abstract unification function associating to two terms either their most general unifier, or the answer that there is none. Application of substitution θ to a term t, and its composition with another substitution ρ will be denoted by the usual postfix notation, $t\theta$, $\theta\rho$.

Renaming of variables in a term t, at the current *renaming level* $vi \in \mathcal{N}$, will be denoted by t' (and accompanied by an update $vi := vi + 1$, to ensure freshness of subsequent renamings).

The above mentioned switching of *modes* will be represented by a distinguished element $mode \in \{Call, Select\}$ indicating the action to be taken at *currnode*: creating the resolvent states, or selecting among them. To be able to speak about *termination* we will use a distinguished element $stop \in \{0, Success, Failure\}$, to indicate respectively running of the system, halting with success and final failure. In a similar manner we shall handle error conditions by a distinguished element *error*, taking values in a set of error messages.

We shall keep the above mentioned notion of *candidate clause* (for executing a literal) abstract (regarding it as implementation defined), assuming only the following integrity constraints: every candidate clause for a given literal

- has the proper predicate (symbol), i.e. the same predicate as the literal (*correctness*); and
- every clause whose head unifies with the given literal is candidate clause for this literal (*completeness*).

One might think of considering any clause occurrence whose head is formed with the given predicate, or the clause occurrences selected by an indexing scheme, or just all occurences of unifying clauses, like in SLD resolution.

In order to allow for *dynamic code* and related operations, we have to speak explicitly about different occurrences of clauses in a program. We hence introduce an abstract universe *CODE* of clause occurrences (or pointers), coming with functions

$$clause : CODE \rightarrow CLAUSE$$
$$cll : NODE \rightarrow CODE$$

where $cll(n)$ is the candidate clause occurrence ("clauseline") corresponding to a candidate son n of a computation state, and $clause(p)$ is the clause "pointed at" by p. Note that we do not assume any ordering on *CODE*. We instead assume an abstract function

$$procdef: LIT \times PROGRAM \rightarrow CODE^* + \{nil\},$$

of which we assume to yield the (properly ordered) list of the candidate clause occurrences for the given literal in the given program (the meaning of *nil* in codomain of *procdef* will come forward in the section on database operations). The current program is represented by a distinguished element *db* of *PROGRAM* (the *database*). Note that existence of *procdef* is all that we assume, for now, of the abstract universe *PROGRAM*.

This concludes the definition of the signature of Prolog tree algebras. Minor additions, pertaining only to some special constructs, will be presented in corresponding sections.

Notationally, we usually suppress the parameter *currnode* by writing simply

$$\begin{aligned} father &\equiv father(currnode)\\ cands &\equiv cands(currnode)\\ s &\equiv s(currnode)\\ decglseq &\equiv decglseq(currnode)\\ fst\_cand &\equiv fst(cands). \end{aligned}$$

Components of a decorated goal sequence will be accessed as

$$\begin{aligned} goal &\equiv fst(fst(decglseq))\\ cutpt &\equiv snd(fst(decglseq))\\ act &\equiv fst(goal)\\ cont &\equiv [\ \langle rest(goal), cutpt\rangle \mid rest(decglseq)\ ] \end{aligned}$$

with *act* standing for the selected literal (*activator*), and *cont* for *continuation*.

We shall also abbreviate

attach $t_1, \ldots, t_n$ **with** *updates* $\equiv$ **extend** *NODE* **by** $t_1, \ldots, t_n$ **with**
father(t_i) := *currnode*
cands := $[\ t_1, \ldots, t_n\ ]$
updates
endextend

3.2. Rules for Core Prolog

Now to dynamics. We assume the following **initialization of Prolog tree algebras**: *root* is supposed to be the nil element — on which no function is defined — and father of *currnode*; the latter has a one element list $[\ \langle query, root\rangle\ ]$ as decorated goal sequence, and empty substitution; the mode is *Call*, *stop* has value 0; *db* has the given program as value. The list *cands* of resolvent states is not (yet) defined at *currnode*.

We now define the four basic rules by which the system attempts, given a pure Prolog program, to reach a state *stop*ping with *Success* (due to first successful execution of the query) or with final *Failure* (by backtracking all the way to *root*). We introduce the following abbreviation for backtracking to father:

backtrack ≡ **if** *father* = *root*
then *stop* := *Failure*
else *currnode* := *father*
mode := *Select*

In case of final *Failure* no transition rule will be applicable, which is a natural notion of "terminating state". All transition rules will thus be tacitly assumed to stand under the guard

OK ≡ *stop* = 0 & *error* = 0,

where *error* is a 0-ary function representing the error condition, cf. Section 7.

The following **query success rule** — for successful halt — will then lead to successful termination when all goals have been executed:

if *decglseq* = [ ]
then *stop* := *Success*

The answer substitution is, of course, represented by the value of s restricted to the variables of the initial query.

The stop rule may be easily modified to allow for the habit of top-level Prolog interpreters to ask the user whether he wants alternative solutions: introduce a 0-ary function *more_solutions_wanted* with an additional rule

if *more_solutions_wanted*
&*stop* = *Success*
then *backtrack*
stop := 0

Since we do not set its value, *more_solutions_wanted* would be an *external* function in the sense of [28].

The following **goal success rule** describes success of a clause body, when the system continues to execute the rest of its goal sequence.

if *goal* = [ ]
then *proceed*

For pure Prolog we are describing in this section, *proceed* can be understood just as

proceed ≡ *decglseq* := *rest*(*decglseq*),

i.e. popping the calling stack. In view of some constructs which, precisely at this moment of completing a call, have to take some special action — indicated by a mark immediately following the procedure body — full meaning of *proceed* is given as

proceed ≡ **if** *rest*(*decglseq*) = [ ] **or** *snd*(*decglseq*) ∉ *MARK*
then *decglseq* := *rest*(*decglseq*)
else *special_action*(*snd*(*decglseq*), *currnode*)

Goal success and query success rules exclude each other; in fact *goal* = [ ] may only be true when *goal* is defined, due to usage of equality in partial algebras. Likewise, the existence of *act*, assumed in rules to follow, excludes the guards of both success rules.

As explained above, an attempt at resolution step is split into *calling* the activator (to create new candidate nodes for alternative resolvents of *currnode*), to be followed by *selecting* one of them. We will correspondingly have two rules. The following **call rule**, invoked by having a user-defined activator in *Call* mode, will create as many sons of *currnode* as there are candidate clauses in the procedure definition of its activator, to each of which the corresponding clause(line) will be associated.

if *is_user_defined*(*act*)
& *mode* = *Call*
then
attach $t_1, \ldots, t_n$ **with**
$cll(t_i) := c_i$
mode := *Select*
where [$c_1, \ldots, c_n$] = *procdef*(*act*, *db*)

where *is_user_defined* is a boolean function recognizing those literals whose predicate symbols are user defined (as opposed to built-in constructs). Note that goals and substitutions, attached to candidate sons, are at this point undefined, and that the value of *currnode* does not change. Note that, in case of dynamic database, this rule commits to the so-called logical view (see Section 5).

The following **selection rule** attempts to select a candidate resolvent state (selecting thereby the associated clause occurrence). If there is none, the system backtracks.

In the other case an attempt to resolve with *fst_cand* is made. If the renamed head of the selected clause does not unify with the activator, the corresponding son is erased from the list of candidates. Otherwise the selected clause is activated: the corresponding son becomes the value of *currnode* in *Call* mode and gets erased from its father's *cands* list. This is represented by the update

go_fst_cand ≡ *currnode* := *fst_cand*
cands := *rest*(*cands*)
mode := *Call*

Remember that these updates, when called within a rule, have to be executed simultaneously.

The decorated goal sequence is defined by executing the resolution step — replacing the activator by clause body (decorated with appropriate cutpoint) and applying the unifying substitution to both *s* and (new) *decglseq*. The current value of *father* gets stored as *cutpt*, since this is the value *father* should resume were *cut* to be executed

within that body, cf. *cut* rule below. Note also the updating of the (just used) value of variable renaming index *vi*. This whole attempt at resolving with *fst_cand* is expressed by the update

resolve ≡ **if** $\theta = nil$
  then $cands := rest(cands)$
  else *go_fst_cand*
    $decglseq(fst\_cand) := new\_decglseq$
    $s(fst\_cand) := s\,\theta$
    $vi := vi + 1$
  where $Hd :- Bdy = clause(cll(fst\_cand))$
    $\theta = mgu(act, Hd')$
    $new\_decglseq = [\ \langle Bdy', father\rangle \mid cont\ ]\ \theta$

With these abbreviations the select rule takes the form

if *is_user_defined*(*act*)
  & *mode* = *Select*
thenif *cands* = [ ]
then *backtrack*
else *resolve*

This concludes the list of rules for our model of pure Prolog. Obviously, our model is *deterministic:* at most one rule is applicable in any given situation, i.e. the guards are, under our conventions, pairwise exclusive. This is only a natural reflection of Prolog (unlike SLD resolution) being a deterministic language.

3.3. Prolog Tree Theorem

It is easy to relate this model, as far as pure Prolog is concerned, to the established body of logic programming theory: the model can be viewed as an incremental construction of (a part of) the SLD-tree.

The dynamic notion of computation tree naturally *classifies* the nodes: a node *n* is

- **visited** if *cands*(*n*) is defined, i.e. if it is, or has already been, the value of *currnode*;
- **active** if it is *currnode* or on the *father* path from *currnode* to *root*;
- **abandoned** if it is visited but not active, i.e. has been backtracked from;

There will in general be other nodes, created as candidates but never visited, which play no role in the computation.

By extracting the goal components of decorated goals and by flattening the lists we obtain the goals of SLD-resolution, i.e. by reading

$$[\ \langle G_1, cutpt_1\rangle, \ldots, \langle G_n, cutpt_n\rangle\ ] \text{ as } flatten([\ G_1, \ldots, G_n\ ])$$

we obtain a correspondence for which the following is true (given a pure Prolog program, i.e. one not containing *cut*):

Lemma 1. *Given a pure Prolog program and a query, every visited node of the Prolog tree corresponds to a node of the SLD-tree (with the same substitution).*

Proof. By induction over time of (number of rule executions preceeding) the first visit. Induction step follows immediately from selection rule together with the definitions of SLD-tree [33] and candidate clause. □

Lemma 2. *Given a pure Prolog program and a query, every abandoned node of the Prolog tree corresponds to a failed node of the SLD-tree.*

Proof. By induction over time of abandonement. A node can namely be abandoned (in pure Prolog) only if, in *Select* mode, either

(a) it has never had any unifying sons, or

(b) all of its unifying sons are already abandoned,

cf. selection rule. Noting that the first node to be abandoned gets abandoned by (a), induction step follows by comparing the definitions. □

The lemmata yield immediately the following theorem, which may be taken as expressing *correctness of Prolog trees wrt SLD resolution*:

Prolog Tree Theorem. *Given a pure Prolog program and a query,*

(i) *if the Prolog tree succeeds, i.e. reaches a state with stop = Success, then SLD resolution succeeds with the same substitution;*

(ii) *if the Prolog tree fails, i.e. reaches a state with stop = Failure, then SLD resolution (finitely) fails.*

Counterexamples to *completeness*, i.e. examples of Prolog following an infinite path in presence of a (finite) successful one, are well known and the reader may simulate them on our Prolog trees. The converse of (i) is thus not true, while the converse of (ii) obviously is.

4. Control constructs and predicates

Here we define, by rules, control constructs of Prolog — in fact all such constructs listed in the draft standard proposal [47]. The first part will treat those constructs which, upon failure, cannot be resatisfied (so-called "deterministic" constructs in Prolog jargon, what is not to be confused with the usual notion, under which *all* constructs of Prolog are deterministic). Resatisfiable ("nondeterministic") constructs, which will be dealt with in the second part, are characterized by a necessity to create alternative candidate nodes. For "deterministic" constructs we shall then create no new nodes at all — they will be executed "in place".

Goal **true** succeeds, **fail** triggers backtracking, while $s = t$ attempts to unify the terms. This is precisely expressed by the rules

```
if act = true                    if act = (s = t)
then succeed                     thenif θ = nil
                                 then backtrack
                                 else decglseq := cont θ
                                      s := s θ
if act = fail                    where θ = mgu(s, t)
then backtrack
```

where *succeed* stands for *decglseq* := *cont*.

The **cut** is usually explained by the metaphor of cutting away a part of the tree, which would, in our framework, amount to recursively resetting the *cands* lists to [ ] all the way from *currnode* to *cutpt*. We shall instead, even more simply, bypass the tree section becoming redundant, by updating *father* to *cutpt*:

```
if act = !
then father := cutpt
     succeed
```

The syntax, allowing arbitrary terms to occur in goals, also allows us to have, in clause bodies, uninstantiated variables in place of literals. Were such a variable, at run time, instantiated to a callable literal, the 4-rules-Prolog of the previous section would execute it without complaining. This is the **metacall** facility of Prolog, also called "textual substitution". In spite of its apparent innocuousness, *metacall* provides Prolog with reflective capabilities — with it it is easy to write an interpreter for Prolog in a few lines of Prolog. A *metacall* automatically inherits the cutpoint of its surrounding body — *metacall*ing a *cut* would cut off, among other things, all alternative clauses for the calling predicate. This property is usually called *transparence* (of *metacall*) for the *cut*.

There is also a **call/1** predicate, explicitly invoking its argument as the activator. The only difference between *call* and *metacall* consists in the former creating its own scope for the *cut* — *call* is not transparent.

```
if act = call(G)
then decglseq := [ ⟨[ G ], father⟩ | cont ]
```

Call could have been equivalently described, relying on *metacall*, by the clause

```
call(X) :- X.
```

It might be a useful excercise to work out the details of *call*(!), and to see in which sense it is equivalent to *true* (although some systems have it differently — cf. discussion in [7, p.55]).

In the Appendix we give the simple rule for the predicate **once**, usually explained as "calling its only argument, without resatisfiability." The rule is equivalent to definition of *once* by the single clause

```
once(X) :- X, !.
```

Note that definition by Prolog clauses here *does not* presuppose Prolog — other than as interpreted by our rules. It is in the presence of rules that such definition acquires the status of a precise mathematical definition.

The notion of transparence for the *cut* is probably understood once it becomes clear that, in this clause, a *call* would have been equivalent to the *metacall*.

In the sequel we shall rely on an abstract syntax, using **and**, **if_then**, or **if_then_else** in order to avoid fussing with syntactical problems.

The call $and(G_1, G_2)$ is usually explained as "calling G_1, G_2 in sequence, succeeding when they both succeed", with the proviso that it is transparent for the *cut*.

if $act = and(G_1, G_2)$
then $decglseq := [\ \langle [\ G_1, G_2\ ], cutpt\rangle \mid cont\ ]$

The call $if\_then(I,T)$ is also executed by executing its arguments in a sequence, but with a very different treatment of cutpoints. Once the first argument succeeds, the call is committed to success of T — if T subsequently fails, the whole call will fail. This behaviour is usually explained as "executing a local *cut*". Further, *if_then* is transparent only for *cut*s in the second argument.

if $act = if\_then(I,T)$
then $decglseq := [\ \langle [\ I, !\ ], father\rangle, \langle [\ T\ ], cutpt\rangle \mid cont\ ]$

Note that this is *not* equivalent to $and(and(I,!),T)$. Note also that, if the guard I fails, the whole call *fails* — very much unlike the meaning of a conditional in other languages for programming or reasoning.

Note that the preceding rules — as well as *goal success* — never mention *mode*. It is however easy to see (by induction) that these rules can only be invoked in *Call* mode — current *decglseq* is namely changed only by rules which preserve the mode, or by *Select* rule, which switches it to *Call*.

The "nondeterministic", i.e. resatisfiable, control constructs, create (a fixed number of) alternative sons, and will thus have to rely on switching of modes. Since the selection of alternatives will not depend on unification, the sons can be fully decorated already in *Call* mode — hence a simplified **selection rule** will suffice.

if $is\_bip(act)$
  & $mode = Select$
thenif $cands = [\ ]$
then *backtrack*
else *go_fst_cand*

We assume here that the Boolean *is_bip* function recognizes exactly those literals whose predicate symbols denote those built-in constructs which do not have a dedicated *Select* rule (like database operations and solution collecting predicates, cf. below).

Given such a uniform selection method, each ("nondeterministic" built-in) construct will be defined by its calling rule.

The relation of *or* to *if_then_else* is similar to that of *and* to *if_then*.

if $act = or(G_1, G_2)$
  & $mode = Call$
then
 attach t_1, t_2 **with**
  $decglseq(t_i) := [\ \langle [\ G_i\ ], cutpt\rangle \mid cont\ ]$
  $s(t_i) := s$
 $mode := Select$
if $act = if\_then\_else(I, T, E)$
  & $mode = Call$
then
 attach t_1, t_2 **with**
  $decglseq(t_1) := [\ \langle [\ I\ ], currnode\rangle, \langle [\ !\ ], father\rangle, \langle [\ T\ ], cutpt\rangle \mid cont\ ]$
  $decglseq(t_2) := [\ \langle [\ E\ ], cutpt\rangle \mid cont\ ]$
  $s(t_i) := s$
 $mode := Select$

Note that *if_then_else*(I, T, E) is *not* equivalent to *or*(*if_then*$(I, T), E)$, in spite of graphical resemblance of rules (and usual Prolog syntax) — just consider *if_then_else*(*true*, *fail*, *true*). The Prolog *if_then_else* is much closer to *if_then_else* than the Prolog *if_then* is to *if_then*.

In the Appendix we give also the rules for **not/1, repeat/0**, which are usually defined by the clauses

```
not(X) :- call(X), !, fail.
not(X).

repeat :- repeat.
repeat.
```

Although the first clause for $not(X)$ is usually written with a *metacall*, it is wrong (if $not(X)$ should succeed when X fails) — consider *not*(*and*(!, *fail*)).

The control constructs, treated so far, influence the flow of Prolog control in a local, gentle way — by setting its boundary conditions — whereas **catch** and **throw** affect it drastically, by allowing an instantaneous jump out of a deeply nested, maybe recursive, call. Their origin in LISP is reflected in their names; *setjmp*, *longjmp* play a very similar role in C. The usual verbal explanation runs roughly as follows: *catch*(*Goal*, *Catcher*, *Recovery*) executes *Goal* in the usual way — as far as within that call no *throw* is executed. If however a *throw*(*Ball*) is executed within this call, the *Ball* seeks the nearest surrounding *catch* with whose *Catcher* it can unify. Upon success *Recovery* is executed, proceeding further from that call of *catch*.

A precise definition is given by the following rules. Note that this is the first time we use the special *marks* we have allowed ourselves to put into the calling stack. The mark, left by executing *catch* in *node*, is of form *Catchpt*(*node*), and is kept in the *decglseq*

immediately behind the goal argument. The information stored in such a mark is *node* — the functor *Catchpt* merely indicates the kind of special action this mark serves for. All *throw* has to do then is seek an appropriate mark, recursively down the *decglseq* — this search procedure is encapsulated in an abstract function called *find_catcher* below.

if $act = catch(G, C, R)$
  & $mode = Call$
then
 attach t **with**
  $decglseq(t)$
   $:= [\langle [\, G\,], currnode\rangle,$
      $Catchpt(currnode) \mid cont\,]$
  $s(t) := s$
  $mode := Select$

if $act = throw(B)$
  & $mode = Call$
  & $find\_catcher(cont, B) \neq root$
then $currnode := found$
    $decglseq(found)$
     $:= [\langle [\, R\,], father(found)\rangle$
        $\mid cont(found)\,]\,\theta$
    $s(found) := s(found)\,\theta$
where $found = find\_catcher(cont, B)$
     $act(found) = catch(G, C, R)$
     $\theta = mgu(B, C)$

where the function

$$find\_catcher : DECGOAL^* \times TERM \rightarrow NODE$$

is defined as satisfying the following requirements.

$$find\_catcher([\,], B) = root$$

$$find\_catcher([\, Entry \mid Rest\,], B) = \textbf{if } Entry = Catchpt(caller) \;\&\; act(caller) = catch(G, C, R) \;\&\; mgu(B, C) \neq nil \textbf{ then } caller \textbf{ else } find\_catcher(Rest, B)$$

Note that, due to the following definition of *special_action* for $Catchpt(node)$ as first parameter

$$special\_action(Catchpt(node), n) \equiv decglseq(n) := rest(rest(decglseq(n)))$$

the marks are placed in such a way that, by the full *proceed* rule of Section 3, normal operation of the system will not be influenced by the existence of *Catchpt* marks — only *throw* sees them.

The reader who knows about implementation, will recognize *find_catcher* as embodying a search of the environment stack. This strategy could be optimized, allowing access to the next *catch* in constant time (cf. [24]), by linking the *Catchpt* marks and storing the top one in a fixed place. While such implementation issues can be sometimes clearly expressed even in this abstract framework, we shall not go into them here (but see [18,16]).

5. Database operations

The understanding of programs as data is pushed in Prolog to the point that even dedicated constructs for explicit viewing and modification of the program, during execution, are provided. Since the rest of Prolog is independent of that possiblity, the signature, and the assumptions we imposed upon it, were so far consistent with seeing the program *db* as constant. It is only here that we shall, in an orthogonal way, add the fragment of signature and the assumptions allowing us to view *db* as variable in time.

Current practice (and the draft standard proposal) classify user defined predicates into *static* and *dynamic*, reflected here by a Boolean recognizer *dynamic*, which we shall, for simplicity, apply to literals (instead of their predicate indicators). The basic operations of inspecting and modifying (by deletion and insertion of clauses which are given in the form (*Head, Body*)) will be represented by functions

$$\begin{aligned}
\mathit{clause_list} &: \mathit{TERM} \times \mathit{TERM} \times \mathit{PROGRAM} \longrightarrow \mathit{CODE}^* + \{\mathit{nil}\} \\
\mathit{predicate_list} &: \mathit{PROGRAM} \longrightarrow \mathit{PI}^* \\
\mathit{delete_cl} &: \mathit{CODE} \times \mathit{PROGRAM} \longrightarrow \mathit{PROGRAM} \\
\mathit{delete_pi} &: \mathit{PI} \times \mathit{PROGRAM} \longrightarrow \mathit{PROGRAM} \\
\mathit{insert} &: \mathit{TERM} \times \mathit{TERM} \times \mathit{PROGRAM} \longrightarrow \mathit{PROGRAM}
\end{aligned}$$

where $PI \subset CODE$ is the universe of predicate indicators, and *insert* is a generic instance of functions *inserta* and *insertz* for inserting at beginning and at end respectively. The following integrity constraints express the mutual dependencies of these functions and *procdef*:

$$p/n \in \mathit{predicate\_list}(db) \Longleftrightarrow (\exists t_1, \ldots, t_n, x)(\mathit{clause\_list}(p(t_1, \ldots, t_n), x, db) \neq \mathit{nil})$$

$$\begin{aligned}
\mathit{clause_list}(g, X, db) &= \mathit{procdef}(g, db) \\
\mathit{clause_list}(x, y, \mathit{delete_cl}(c, db)) &\not\ni c \quad \text{for any } x, y \\
\mathit{predicate_list}(\mathit{delete_pi}(p/n, db)) &\not\ni p/n \\
\mathit{clause_list}(H, B, \mathit{inserta}(H, B, db)) &= [\, c \mid \mathit{clause_list}(H, B, db) \,] \\
&\qquad \text{iff } \mathit{clause}(c) = H :\!- B
\end{aligned}$$

with a similar condition for *insertz*. The *assert* predicate, coming in 2 versions, **asserta, assertz**, coupled to corresponding *insert* functions, appends a clause — to be found by subsequent calls — at the appropriate end of the procedure. Note how this is forced by conditions relating *insert*, *clause_list* and *procdef*.

if $act = \mathit{assert}(H, B)$
  & $\mathit{dynamic}(H)$
then $db := \mathit{insert}(H, B, db)$
  succeed

Whereas *assert* is "deterministic", the predicates **clause** and **retract**, which serve for viewing and deleting, respectively, of alternative clauses matching their arguments, are resatisfiable. Since their applicability to a clause depends on successful unification, they can be described following closely the pattern of *Call* and *Select* rules for user-defined predicates of Section 3.

if $act = clause(H, B) \mid retract(H, B)$
  & $dynamic(H)$
  & $mode = Call$
then attach $[\, t_1, \ldots, t_n \,]$
  with $cll(t_i) = c_i$
  $mode := Select$
where $clause\_list(H, B, db) = [\, c_1, \ldots, c_n \,]$

if $act = clause(H, B) \mid retract(H, B)$
  & $dynamic(H)$
  & $mode = Select$
thenif $cands = [\,]$
then *backtrack*
else *resolve*
  if $\theta \neq nil$
  then $\_ \mid db := delete\_cl(cll(fst\_cand), db)$
where
  $\theta = mgu(H' :- B', clause(cll(fst\_cand)))$
  $new\_decglseq = cont\,\theta$

where $\_ \mid \_$ is used as obvious shorthand for two similar rules. In fact, these rules may be obtained from those of Section 3 by a straightforward transformation, where *resolve* comes with redefined $\theta, new\_decglseq$.

The constructs **current_predicate** and **abolish** are isomorphic to the *clause*, *retract* pair, with a predicate indicator instead of either a clause or a code pointer (clauseline), *predicate_list* instead of *clause_list*, and *delete_pi* instead of *delete_cl*, cf. the Appendix. With the benefit of hindsight one can see that it is due to decomposing the basic Prolog computation step into calling and selecting that the isomorphism of user-defined predicates to *clause, retract, current_predicate, abolish* becomes explicit. Note that the function *predicate_list* defines the order in which predicate indicators are found by *current_predicate*; it may be considered as implementation dependent, as required by [47]. Given the constraints, our rules do not specify any action in the case access to a clause for an *abolish*ed predicate is attempted — this falls under error handling, and will be discussed in Section 7.

The interpretation our rules give to database operations is that of so-called *logical view* [32] — modifications of the database affect only subsequent calls, but the alternatives for current call remain as they were at time of call, by being copied into the tree structure by the appropriate call rule. The logical view is the one adopted in the draft standard proposal — for an analysis of alternative possibilities see [11,16]. Here it will have to suffice to say that the following program

$q :- assertz(q), fail.$
$r :- retract(r, X), fail.$
$r.$

differentiates between different views, and unfolds the related difficulties [16]. Under the logical view q fails while r succeeds — the reader may work it out with the rules.

6. Solution collecting predicates

The predicates *findall, bagof, setof* bring the second-order notion of comprehension (collection) into the first-order world of Prolog. The first-order realizations of the second-order comprehension principle are necessarily approximate and imperfect — which partially accounts for some difficulties a specification of these predicates must encounter.

A call of *findall*(*Term*, *Goal*, *Bag*) finds values of *Term* as instantiated by answer substitutions θ of all solutions of *Goal*, collects them into a list (in the ordering of solutions), and unifies this list with *Bag*. The independence of solutions is reflected by fresh renaming of all uninstantiated variables which might occur in *Term* θ, at every collecting step.

A realization of **findall**, if it is to use backtracking to make Prolog find all solutions automatically, must introduce some external mechanism, to remember (the needed fragment of) the computed answer substitution, which Prolog alone would forget on backtracking. The external mechanism, or special action to be taken immediately after completion of every successful computation of *Goal*, is in our rules indicated by a mark *Collect*(*caller*). The special action consists in collecting, at *caller*, the renamed *Term* θ, and triggering backtracking. The solutions are accumulated in a list provided by an additional (partial) function

$$term\_list : NODE \rightarrow TERM^*$$

Insertion of the mark, and starting computation of *Goal* from a newly created node, is done in *Call* mode. We also use a *Select* mode, this time not to pass to alternatives, since there are none, but to complete the process when backtracking finally leaves *Goal*, by unifying *Bag* with *term_list*.

Since *Call* rule is common to *findall, bagof, setof*, we shall list it once, with the generic predicate name SETEXPR standing for each of the three.

if *act* = SETEXPR(*T*, *G*, *B*)
  & *mode* = *Call*
then
 cands := [ ]
 term_list(*currnode*) := [ ]
 extend *NODE by t* **with**
  father(*t*) := *currnode*
  decglseq(*t*) := [⟨[*G*], *currnode*⟩,
                Collect(*currnode*)]
  s(*t*) := *s*
  currnode := *t*
 endextend

if *act* = *findall*(*T*, *G*, *B*)
  & *mode* = *Select*
thenif ρ = *nil*
then *backtrack*
else *s* := *s* ρ
    decglseq := *cont* ρ
    mode := *Call*
where
 ρ := *mgu*(*term_list*(*currnode*), *B*)

$special\_action(Collect(caller), node) \equiv$ **if** $act(caller) = findall(T,G,B)$
then let $\theta = s(node)$
$term\_list(caller) :=$
$append(term\_list(caller), [\ (T\theta)'\ ])$
$vi := vi + 1$
$backtrack$

The **bagof**, **setof** predicates do everything *findall* does, *classifying* also the solutions according to the values taken by *parameters*, i.e. those variables which occur in *Goal* but not in *Term*. The list of solutions, i.e. values of $Term\,\theta$ which come together, according to this classification, is unified with *Bag* as one (alternative) solution to *bagof*. It follows that in case of no solution *bagof* should fail, unlike *findall*, since there are no parameter values a solution could correspond to. It also follows that, in case of no parameters, *bagof* should be the same as *findall* (given that there is a solution), since there is nothing according to what one could classify.

The (rough) idea may be seen from the following example.

If we had a database of $child(peter,fred)$, $child(paul,fred)$, $child(mary,joan)$, $child(fred,ann)$, $child(joan,ann)$, the solution to grandchildren collecting call

$$findall(X,\ and(child(X,Parent), child(Parent,ann)),\ Grandchildren)$$

would be $Grandchildren = [\ peter, paul, mary\ ]$ (without binding *Parent*), while a call

$$bagof(X,\ and(child(X,Parent), child(Parent,ann)),\ Grandchildren)$$

would have *two* solutions,

$$Parent = fred,\ Grandchildren = [\ peter, paul\ ]$$
$$Parent = joan,\ Grandchildren = [\ mary\ ]$$

The common core of solution collecting predicates is reflected in having a common *Call* rule, as well as in the same structure of *special action*, cf. below, except that, instead of just appending, the new contribution is *put in* its *place* according to the computed value of parameters $\vec{Y}$. The *Select* rule below has to have alternative sons, in view of resatisfiability of *bagof, setof*; additional unification affects the final binding of parameters $\vec{Y}$ to their computed value.

How a computed T with parameters $\vec{Y}$ is to be put in its place, depends on the classification principle, which is, in the definition of *put_in_place* update below, represented by a function

$$find\_class : TERM^* \times NODE^* \rightarrow NODE + \{nil\}$$

Of this function we assume to, for given parameters $\vec{Y}$ and a list of candidate nodes, select a node in which term-parameters pairs $T_i - \vec{Y}_i$, belonging to the same class with $\vec{Y}$, are already being collected. If one is found, the $T_i - \vec{Y}_i$ entry is appended to the list, unifying thereby the parameters to ensure their identity. The classification principle must

of course be such as to make this unification coherent. If no such node exists, *find_class* will return *nil*, and *put_in_place* will create a new son, insert it into the list of those corresponding to already existing classes, and start collecting a new class there.

The predicate *setof* is usually explained as being the same as *bagof*, with, additionally, sorting the solutions, removing thereby any duplicates. For this purpose an additional sorting function is used.

In the *Select* rule below, we shall use a function *params*, which extracts a list of all variables occurring in the *Goal* but not in *Term*. The syntax also allows *quantified goals* as arguments to set expressions, where some variables are explicitly excluded from being parameters. We assume *params* to know that.

if $act = bagof(T, G, B) \mid setof(T, G, B)$
  & $mode = Select$
thenif $cands = [\,]$
then *backtrack*
elsif $\rho = nil$
then $cands := rest(cands)$
else *go_fst_cand*
  $s(fst\_cand) := s(fst\_cand)\ \rho$
  $decglseq(fst\_cand) := cont(fst\_cand)\ \rho$
  $mode := Call$
where
$\vec{Y} = params(T, G)$
$term\_list(fst\_cand) = [\,T_1 - \vec{Y}_1, \ldots, T_n - \vec{Y}_n\,]$
$\theta = mgu(\vec{Y}, \vec{Y}_n)$
$\rho = mgu([\,T_1, \ldots, T_n\,]\,\theta, B) \mid mgu(sort([\,T_1, \ldots, T_n\,]\,\theta), B)$

$special\_action(Collect(caller), node) \equiv$ **if** $act(caller) = bagof(T, G, B)$
  or $act(caller) = setof(T, G, B)$
then $put\_in\_place((T\theta)', (\vec{Y}\theta)', caller)$
$vi := vi + 1$
backtrack
where $\vec{Y} = params(T, G)$
  $\theta = s(node)$

$put\_in\_place(T, \vec{Y}, caller) \equiv$ **if** $class = nil$
then extend *NODE* **by** t **with**
  $father(t) := caller$
  $cands(caller) := ins(t, cands(caller))$
  $term\_list(t) := [\,T - \vec{Y}\,]$
  $s(t) := [\,]$

else
term_list(*class*)
:= *append*([$T_1 - \vec{Y}_1, \ldots, T_m - \vec{Y}_m$],
[$T - \vec{Y}$]) θ
s(*class*) := *s*(*class*) θ
where *class* = *find_class*($\vec{Y}$, *cands*(*caller*))
term_list(*class*) = [$T_1 - \vec{Y}_1, \ldots, T_m - \vec{Y}_m$]
$\theta = mgu(\vec{Y}, \vec{Y}_m)$

Disagreements in specification and implementation of these predicates can all be viewed as encapsulated in classification and sorting principles involved, i.e., in the context of above rules, in the *find_class, sort, insert* functions. It is indeed disputable whether the proposed (or any) classifying and sorting principles are sound when parameters are not fully instantiated. Therefore we leave these functions abstract, but see [18] for analysis, rationale and criticism.

7. Error handling

Error handling, as prescribed by the draft standard proposal, is very easy to describe in this framework. We have, in Section 3, already mentioned putting all "normal" rules under the guard *error* = 0. Exceptional rules, without that guard, will be *error handlers*. Error handling, as foreseen by the standard proposal, is very uniform — *throw* an error indicator — enabling the programmer to write his own error handlers. The only exception is *system_error*, in which case the action is "implementation dependent". Thus the guards of *all* our rules do exclude the condition *error* = *system_error* — in that case the system would just halt. All other errors would then be handled by the following uniform rule:

if *error* $\notin \{0, system\_error\}$
then *decglseq* := [$\langle throw(rep(error)), . \rangle$ | *cont*]
error := 0

Of function *rep* here we assume to represent the error indicator by a Prolog term. Note that *error* has to be reset, so that a *catch* can continue the computation. Since *throw* does not use the cutpoint, we have put a dot instead. We could not have skipped the continuation — it is essential to *throw*.

The orthogonality of normal operation and error handling can be expressed by composition operators on evolving algebras introduced in [26]: if the set of all "normal" rules, including those below, is denoted as *Prolog*, and the (singleton containing) *error* rule above as *Error*, Prolog with error handling will be represented by the evolving algebra

(*OK*?*Prolog* | *Error*)

where $\phi ?A$ denotes evolving algebra A with guard ϕ added to all its rules, and _ | _ denotes disjoint sum of rule sets.

The default error behaviour, if the programmer has not encapsulated his code in appropriate calls of *catch*, would then be given by the rule

if *act* = *throw*(*B*)
  & *mode* = *Call*
  & *find_catcher*(*cont*, *B*) = *root*
then *error* := *system_error*,

cf. rule for *throw*. The rest is a description of conditions under which error situations occur. For instance, an uninstantiated activator in *Call* mode produces an instantiation error, while an instantiated activator which is not a callable predication produces a type error.

if *is_var*(*act*)
  & *mode* = *Call*
then *error* := *instantiation_error*

if *wrong_type*(*act*)
  & *mode* = *Call*
then *error* := *type_error*

where *is_var*, *wrong_type* are obvious recognizers — *wrong_type* takes the value *true* precisely on those terms which are neither variables nor literals. The result of an attempt to call a syntactically legal user-defined activator, which is however not in *predicate_list* (cf. Section 5), should, according to standard proposal [47], depend on the value of *undefined_predicate* flag:

if *is_user_defined*(*act*)
  & *mode* = *Call*
  & *procdef*(*act*, *db*) = *nil*
thenif *undefined_predicate* = *error*
then *error* := *undefined_predicate_error*
else *backtrack*
  if *undefined_predicate* = *warning*
  then *warn*

where update *warn* abstractly represents the action of warning the user.

A further typical example would be

if *act* = *clause*(*H*, *B*) | *retract*(*H*, *B*)
  & *mode* = *Call*
thenif *is_var*(*H*)
then *error* := *instantiation_error*
elsif *wrong_type*(*H*)
then *error* := *type_error*
elsif ¬*dynamic*(*H*)
then *error* := *database_error*

etc. etc. All other error conditions, listed in [47], follow this straightforward pattern.

8. Box model

Byrd's box model [41] for debugging is usually explained along the following lines. When a goal is called, a box is created, which is immediately entered via its *Call* port. If the call is completed successfully, the box is left via its *Exit* port. If backtracking later forces an attempt to resatisfy that call, the box is reentered via its *Redo* port. If all attempts at (re)satisfaction fail, the box is finally left via its *Fail* port. Passing through ports, the debugger reports the port and the goal which owns the box (instantiated with the current substitution). The debugger may be switched on and off, and applied to selected user-defined predicates, during the computation.

In order to incorporate the box model, we introduce a 0-ary function *debugging*, and a unary *spying*, indicating whether and which predicates are being debugged. We further insert into our basic rules, at appropriate places, updates of form $say(Port, goal)$, which abstractly represent the reporting.

The *Call* and *Select* rule are then enriched to take the form

```
if is_user_defined(act)
  & mode = Call
then if debugging
       & spying(act)
     then say(Call, act)
     ...
```

```
if is_user_defined(act)
  & mode = Select
thenif cands = [ ]
then backtrack
     if debugging
       & spying(act)
     then say(Fail, act)
else resolve
```

where ... stands for the rest of the *Call* rule, as in Section 3. Reporting of *Redo* will be taken care of by backtracking, which must check the *father* it revisits, being thus

```
backtrack ≡ if father = root
            then stop := Failure
            else currnode := father
                 mode := Select
                 if debugging & spying(act(father))
                 then say(Redo, act(father))
```

In order to record which box is to be exited, we use again the technique of marks, to be pushed for spied activators. Formally, this turns *new_decglseq*, to be set up at selection, to

```
new_decglseq ≡ if spying(act)
               then [ ⟨Bdy', father⟩, Exit(currnode) | cont ] θ
               else [ ⟨Bdy', father⟩ | cont ] θ
```

$special\_action(Exit(node), n) \equiv decglseq(n) := rest(decglseq(n))$
if *debugging* & $spying(act(node))$
then $say(Exit, act(node))$

Usually predicates $debug/0$, $spy/1$ are provided to control the debugger at runtime. Their rules take the form

if act = WHAT | *no*WHAT
then WHAT*ing* := *true* | *false*
succeed

with the obvious range for WHAT.

9. Concluding remarks

We hope to have convinced the reader that we have provided what was promised in the introduction — a simple but rigorous definition, as laid down for reference in the Appendix, which naturally reflects the basic intuitions of full Prolog.

The definition is of course operational: it is presented as an evolving algebra interpreter for full Prolog programs. Nevertheless, it is a *logical* characterization of the language, in at least two ways.

- The four rules, which determine the meaning of pure Prolog, are nothing but the SLD-resolution rule, presented so as to fit the view of Horn-clause sets as *programs*. The appearance of the two success rules reflects the fact that programs consist of procedures, and may terminate — goal success represents exit from a procedure, and query success termination. The decomposition of a resolution step into call and select is one small concession proof theory has to make here to the fact that what has to be interpreted is a *programming language*, and not just a deductive system. This decomposition namely enables smooth and uniform transition from user-defined predicates to resatisfiable built-in constructs. This may help to extend also the established body of the logic programming theory from Horn-clause logic to the real programming language.
- There are strong grounds to maintain that the whole underlying framework of evolving algebras is logical in its essence — its semantics is the standard semantics of logic, structures interpreting a signature, as indicated in the Introduction (but see also [28,26]).

It is thus not accidental that assuming the core of the present model as a starting point has helped to reveal the *logical structure* of the WAM and some of its extension ([17,5,19]). We hope that the full model will serve well for mathematical analysis of real Prolog programs.

Acknowledgements

We thank two anonymous referees for helpful suggestions and criticism. Special thanks to Yuri Gurevich, for permanent discussion of the whole evolving algebra approach and of several early versions of this paper. Last but not least, our thanks go to Informatikzentrum Schloß Dagstuhl, which has provided a splendid environment for our work on the final revision of the paper.

Appendix

For reference, we list here all the universes, functions, abbreviations and rules used, except for usual ones pertaining to Prolog syntax.

A.1. Universes, functions, actions

universes

$$LIT,\ TERM,\ GOAL = TERM^*,\ CLAUSE,$$
$$SUBST,\ MARK,\ PROGRAM,\ ERROR$$

dynamic universe $NODE$, $DECGOAL = (GOAL \times NODE)^*$

functions

$$father : NODE \rightarrow NODE$$
$$root, currnode \in NODE$$

$$decglseq : NODE \rightarrow (DECGOAL + MARK)^*$$
$$s : NODE \rightarrow SUBST$$
$$cll : NODE \rightarrow CODE$$
$$vi \subset N$$

$$cands : NODE \rightarrow CODE^*$$
$$clause : CODE \rightarrow CLAUSE + \{nil\}$$
$$db \in PROGRAM$$
$$procdef : LIT \times PROGRAM \rightarrow CODE^* + \{nil\}$$

$$mode \in \{Call, Select\}$$
$$stop \in \{0, Success, Failure\}$$
$$error \in ERROR$$

abbreviations — functions

$$father \equiv father(currnode)$$
$$cands \equiv cands(currnode)$$

$s \equiv s(currnode)$

$decglseq \equiv decglseq(currnode)$

$fst\_cand \equiv fst(cands)$

$goal \equiv fst(fst(decglseq))$

$cutpt \equiv snd(fst(decglseq))$

$act \equiv fst(goal)$

$cont \equiv [\ \langle rest(goal), cutpt \rangle \mid rest(decglseq)\ ]$

$\mathrm{OK} \equiv stop = 0\ \&\ error = 0$

$lit' \equiv rename(lit, vi)$

abbreviations — updates

$go\_fst\_cand \equiv$ $currnode := fst\_cand$
$cands := rest(cands)$
$mode := Call$

attach $t_1, \ldots, t_n \equiv$ **extend** *NODE*
with *updates* **by** $t_1, \ldots, t_n$ **with**
$father(t_i) := currnode$
$cands := [\ t_1, \ldots, t_n\ ]$
updates

$backtrack \equiv$ **if** $father = root$
then $stop := Failure$
else $currnode := father$
$mode := Select$

$succeed \equiv decglseq := cont$

$proceed \equiv$ **if** $rest(decglseq) = [\ ]$
or $snd(decglseq) \notin MARK$
then $decglseq := rest(decglseq)$
else $special\_action(snd(decglseq), currnode)$

resolution step

$resolve \equiv$ **if** $\theta = nil$
then
$cands := rest(cands)$
else
go_fst_cand
$decglseq(fst\_cand) := new\_decglseq$
$s(fst\_cand) := s\,\theta$
$vi := vi + 1$
where
$Hd :- Bdy = clause(cll(fst\_cand))$
$\theta = mgu(act, Hd')$
$new\_decglseq = [\ \langle Bdy', father \rangle \mid cont\ ]\ \theta$

A.2. Pure Prolog

if *decglseq* = [ ]
then *stop* := *Success*

if *goal* = [ ]
then *proceed*

if *is_user_defined*(*act*)
  & *mode* = *Call*
then
 attach $t_1, \ldots, t_n$
 with $cll(t_i) := c_i$
 mode := *Select*
where $[\, c_1, \ldots, c_n \,] = procdef(act, db)$

if *is_user_defined*(*act*)
  & *mode* = *Select*
thenif *cands* = [ ]
then *backtrack*
else *resolve*

A.3. Control

if *act* = *true*
then *succeed*

if *act* = *fail*
then *backtrack*

if *act* = !
then
 father := *cutpt*
 succeed

if *act* = *call*(*G*)
then *decglseq* := [⟨[*G*], *father*⟩ | *cont*]

if *act* = *once*(*G*)
then *decglseq* := [⟨[*G*, !], *father*⟩ | *cont*]

if *act* = *and*(G_1, G_2)
then *decglseq* := [⟨[G_1, G_2], *cutpt*⟩ | *cont*]

if *act* = *if_then*(*I*, *T*)
then *decglseq* := [⟨[*I*, !], *father*⟩,
 ⟨[*T*], *cutpt*⟩ | *cont*]

if *act* = ($s = t$)
thenif $\theta = nil$
then *backtrack*
else *decglseq* := *cont*θ
  $s := s\theta$
where $\theta = mgu(s, t)$

if *is_bip*(*act*)
  & *mode* = *Select*
thenif *cands* = [ ]
then *backtrack*
else *go_fst_cand*

if *act* = *or*(G_1, G_2)
  & *mode* = *Call*
then
 attach t_1, t_2 **with**
  $decglseq(t_i)$:= [⟨[G_i], *cutpt*⟩ | *cont*]
  $s(t_i) := s$
 mode := *Select*

if *act* = *if_then_else*(*I*, *T*, *E*)
  & *mode* = *Call*
then
 attach t_1, t_2 **with**
  $decglseq(t_1)$:= [⟨[*I*], *currnode*⟩,
   ⟨[!], *father*⟩,
   ⟨[*T*], *cutpt*⟩ | *cont*]
  $decglseq(t_2)$:= [⟨[*E*], *cutpt*⟩ | *cont*]
  $s(t_i) := s$
 mode := *Select*

if $act = not(G)$
 & $mode = Call$
then
 attach t_1, t_2 **with**
  $decglseq(t_1) := [\langle [\, G\,], currnode\rangle, \langle [\, !, fail\,], father\rangle\,]$
  $decglseq(t_2) := cont$
  $s(t_i) := s$
 $mode := Select$

if $act = repeat$
 & $mode = Call$
then
 attach t_1, t_2 **with**
  $decglseq(t_1) := cont$
  $decglseq(t_2) := [\, \langle repeat, .\rangle \mid cont\,]$
  $s(t_i) := s$
 $mode := Select$

catch and throw

$$Catchpt(node) \in MARK$$
$$find\_catcher : DECGOAL^* \times TERM \rightarrow NODE$$

$find\_catcher([\,], B) = root$
$find\_catcher([\, Entry \mid Rest\,], B)$
 = **if** $Entry = Catchpt(caller)$
  & $act(caller) = catch(G, C, R)$
  & $mgu(B, C) \neq nil$
 then $caller$
 else $find\_catcher(Rest, B)$

$special\_action(Catchpt(node), n) \equiv decglseq(n) := rest(rest(decglseq(n)))$

if $act = catch(G, C, R)$
 & $mode = Call$
then
 attach t **with**
  $decglseq(t) := [\langle [\, G\,], currnode\rangle, Catchpt(currnode) \mid cont\,]$
  $s(t) := s$
 $mode := Select$

if $act = throw(B)$
 & $mode = Call$
 & $find\_catcher(cont, B) \neq root$
then $currnode := found$
  $decglseq(found)$
   $:= [\langle [\, R\,], father(found)\rangle \mid cont(found)\,]\,\theta$
  $s(found) := s(found)\,\theta$
where $found = find\_catcher(cont, B)$
  $act(found) = catch(G, C, R)$
  $\theta = mgu(B, C)$

A.4. Dynamic database

predicate indicators $PI \subset CODE$

$$clause\_list : TERM^2 \times PROGRAM \rightarrow CODE^* + \{nil\}$$
$$predicate\_list : PROGRAM \longrightarrow PI^*$$
$$del\_cl : CODE \times PROGRAM \rightarrow PROGRAM$$
$$del\_pi : PI \times PROGRAM \rightarrow PROGRAM$$
$$insert : TERM^2 \times PROGRAM \rightarrow PROGRAM$$

$$p/n \in predicate\_list(db) \Longleftrightarrow (\exists \vec{t}, x)(clause\_list(p(\vec{t}), x, db) \neq nil)$$

$$clause\_list(g, X, db) = procdef(g, db)$$

$$clause\_list(x, y, del\_cl(c, db)) \not\ni c$$

$$predicate\_list(del\_pi(p/n, db)) \not\ni p/n$$

$$clause\_list(H, B, inserta(H, B, db)) = [\ c \mid clause\_list(H, B, db)\ ]$$

$$clause\_list(H, B, insertz(H, B, db)) = append(clause\_list(H, B, db), [\ c\ ])$$

$$clause(c) = H :- B$$

if $act = assert(H, B)$
  & $dynamic(H)$
then $db := ins(H, B, db)$
  $succeed$

if $act = clause(H, B) \mid retract(H, B)$
  & $dynamic(H)$
  & $mode = Call$
  & $clause\_list(H, B, db) = [\ c_1, \ldots, c_n\ ]$
then attach $[\ t_1, \ldots, t_n\ ]$
  with $cll(t_i) = c_i$
  $mode := Select$

if $act = clause(H, B) \mid retract(H, B)$
  & $dynamic(H)$
  & $mode = Select$
thenif $cands = [\ ]$
then *backtrack*
else *resolve*
  if $\theta \neq nil$
  then $\_ \mid db := del\_cl(cll(fst\_cand), db)$
where
  $\theta = mgu(H' :- B', clause(cll(fst\_cand)))$
  $new\_decglseq = cont\,\theta$

if $act = current\_predicate(P) \mid abolish(P)$
  & $mode = Call$
  & $predicate\_list(P, db) = [\ p_1, \ldots, p_n\ ]$
  & $\_ \mid dynamic(P)$
then attach $[\ t_1, \ldots, t_n\ ]$
  with $cll(t_i) = p_i$
  $mode := Select$

if $act = current\_predicate(P) \mid abolish(P)$
  & $mode = Select$
thenif $cands = [\ ]$
then *backtrack*
else *resolve*
  if $\theta \neq nil$
  then $\_ \mid db := del\_pi(cll(fst\_cand), db)$
where $\theta = mgu(P, cll(fst\_cand))$
  $new\_decglseq = cont\,\theta$

A.5. All solutions

$$term\_list : NODE \to TERM^*$$

$$sort : TERM^* \to TERM^*$$

$$find\_class : TERM^* \times NODE^* \to NODE + \{nil\}$$

$$Collect(node) \in MARK$$

if $act = \text{SETEXPR}(T, G, B)$
  & $mode = Call$
then
 $cands := [\,]$
 $term\_list(currnode) := [\,]$
 extend *NODE by* t **with**
  $father(t) := currnode$
  $decglseq(t) := [\langle [\,G\,], currnode\rangle, Collect(currnode)\,]$
  $s(t) := s$
  $currnode := t$

if $act = findall(T, G, B)$
  & $mode = Select$
thenif $\theta = nil$
then *backtrack*
else $s := s\,\theta$
  $decglseq := cont\,\theta$
  $mode := Call$
where $\theta = mgu(term\_list(currnode), B)$

if $act = bagof(T, G, B) \mid setof(T, G, B)$
  & $mode = Select$
thenif $cands = [\,]$
then *backtrack*
elsif $\rho = nil$
then $cands := rest(cands)$
else go_fst_cand
  $s(fst\_cand) := s(fst\_cand)\,\rho$
  $decglseq(fst\_cand) := cont(fst\_cand)\,\rho$
  $mode := Call$
where
 $\vec{Y} = params(T, G)$
 $term\_list(fst\_cand) = [\,T_1 - \vec{Y}_1, \ldots, T_n - \vec{Y}_n\,]$
 $\theta = mgu(\vec{Y}, \vec{Y}_n)$
 $\rho = mgu([\,T_1, \ldots, T_n\,]\,\theta, B) \mid mgu(sort([\,T_1, \ldots, T_n\,]\,\theta), B)$

$special\_action(Collect(caller), node) \equiv$ **if** $act(caller) = findall(T, G, B)$
  then $term\_list(caller) := append(term\_list(caller), [\,(T\,\theta)'\,])$
  elsif $act(caller) = bagof(T, G, B)$
    or $act(caller) = setof(T, G, B)$
  then $put\_in\_place((T\theta)', (\vec{Y}\theta)', caller)$
  $vi := vi + 1$
  backtrack
  where $\vec{Y} = params(T, G)$
    $\theta = s(node)$

$put\_in\_place(T, \vec{Y}, caller) \equiv$ **if** $class = nil$
then extend *NODE* **by** t **with**
$father(t) := caller$
$cands(caller) := ins(t, cands(caller))$
$term\_list(t) := [\, T - \vec{Y} \,]$
$s(t) := [\,]$
else
$term\_list(class)$
$:= append([\, T_1 - \vec{Y}_1, \ldots, T_m - \vec{Y}_m \,],$
$[\, T - \vec{Y} \,])\, \theta$
$s(class) := s(class)\, \theta$
where $class = find\_class(\vec{Y}, cands(caller))$
$term\_list(class) = [\, T_1 - \vec{Y}_1, \ldots, T_m - \vec{Y}_m \,]$
$\theta = mgu(\vec{Y}, \vec{Y}_m)$

A.6. Error Handling

evolving algebra

$(OK?Prolog \mid Error)$

where *Prolog* contains all rules except for *Error* below

$error \in ERROR$
$rep : ERROR \rightarrow TERM$

if *ERROR_CONDITION*
then $error :=$ *ERROR_INDICATOR*

if $act = throw(B)$
& $mode = Call$
& $find\_catcher(cont, B) = root$
then $error := system\_error$

Error

if $error \notin \{0, system\_error\}$
then $decglseq := [\, \langle throw(rep(error)), . \rangle$
$\mid cont \,]$
$error := 0$

A.7. Box model debugger

$debugging \in BOOL$
$spying : PI \rightarrow BOOL$
$Exit(node) \in MARK$
abstract update $say(\_, \_)$

modified call and select rules

```
if is_user_defined(act)
  & mode = Call
then if debugging
       & spying(act)
     then say(Call, act)
     attach t_1, ..., t_n with
       cll(t_i) := c_i
     mode := Select
where [c_1, ..., c_n] = procdef(act, db)
```

```
if is_user_defined(act)
  & mode = Select
thenif cands = [ ]
then backtrack
     ifdebugging
       & spying(act)
     then say(Fail, act)
else resolve
```

modified updates

```
backtrack ≡ if father = root
            then stop := Failure
            else currnode := father
                 mode := Select
                 if debugging & spying(act(father))
                 then say(Redo, act(father))
```

```
new_decglseq ≡ if spying(act)
               then [ ⟨Bdy', father⟩, Exit(currnode) | cont ] θ
               else [ ⟨Bdy', father⟩ | cont ] θ
```

```
special_action(Exit(node), n) ≡ decglseq(n) := rest(decglseq(n))
                                 say(Exit, act(node))
```

```
if act = WHAT | noWHAT
then WHATing := true | false
     succeed
```

```
WHAT ∈ {debug, spy(p/n)}
```

References

[1] H. Aït-Kaci, *Warren's Abstract Machine. A Tutorial Reconstruction* (MIT Press, Cambridge, MA, 1991).

[2] J.H. Andrews, The logical structure of sequential Prolog, University of Edinburgh, Department of Computer Science, Report ECS-LFCS-90-110, 1–37.

[3] K. Apt, Logic programing, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B* (Elsevier, Amsterdam, 1990) 493–574.

[4] B. Arbab and D.M. Berry, Operational and denotational semantics of Prolog, *J. Logic Programming* **4** (1987) 309–329.

[5] C. Beierle and E. Börger, Correctness proof for the WAM with types, in: E. Börger, G. Jäger, H. Kleine Büning and M. Richter, eds., *Computer Science Logic*, Lecture Notes in Computer Science, Vol. 626 (Springer, Berlin, 1992) 15–34.

[6] D. Bjørner and H. Langmaack, Foreword to: D. Bjørner, C.A.R. Hoare and H. Langmaack, eds., *VDM '90. VDM and Z—Formal Methods in Software Development*, Lecture Notes in Computer Science, Vol. 428 (Springer, Berlin, 1990).

[7] E. Börger, A logical operational semantics of full Prolog: Part 1. Selection core and control, in: E. Börger, H. Kleine Büning and M. Richter, eds., *CSL '89, 3rd Workshop on Computer Science Logic*, Lecture Notes in Computer Science, Vol. 440 (Springer, Berlin, 1990) 36–64.

[8] E. Börger, A logical operational semantics of full Prolog: Part 2. Built-in predicates for database manipulations, in: B. Rovan, ed., *Mathematical Foundations of Computer Science 1990*, Lecture Notes in Computer Science, Vol. 452 (Springer, Berlin, 1990) 1–14.

[9] E. Börger, A logical operational semantics of full Prolog: Part 3. Built-in predicates for files, terms, arithmetic and input-output, in: Y.N. Moschovakis, ed., *Logic from Computer Science*, MSRI Publications, Vol. 21 (Springer, Berlin, 1992) 17–50.

[10] E. Börger, F.J. Lopez-Fraguas and M. Rodrigues-Artalejo, A model for mathematical analysis of functional logic programs and their implementations, in: B. Pehrson and I Simon, eds., *Proc. 13th World Computer Congress '94, Vol. 1* (North-Holland, Amsterdam, 1994) 410–415.

[11] E. Börger and B. Demoen, A framework to specify database update views for Prolog, in: J. Małuszynski and M. Wirsing, eds., *Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, Vol. 528 (Springer, Berlin, 1991) 147–158.

[12] E. Börger, I. Durdanovic and D. Rosenzweig, Occam: Specification and compiler correctness. Part I: The primary model, in: E.-R.Olderog, ed., *Proc. Procomet '94, IFIP TC2 Working Conf. on Programming Concepts* (North-Holland, Amsterdam, 1994) 489–508.

[13] E. Börger and E. Riccobene, A formal specification of Parlog, in: M. Droste and Y. Gurevich, eds., *Semantics of programming languages and model theory*, (Gordon & Breach, New York, 1993).

[14] E. Börger and E. Riccobene, A mathematical model of concurrent Prolog, CSTR-92-15, Dept. of Computer Science, University of Bristol, 1992.

[15] E. Börger and D. Rosenzweig, A formal specification of Prolog by tree algebras, in: V. Čerić et al., eds., *Proc. 3rd Internat. Conf. on Information Technology Interfaces* (SRCE, Zagreb, 1991) 513–518.

[16] E. Börger and D. Rosenzweig An analysis of Prolog database views and their uniform implementation, CSE-TR-88-91, University of Michigan, Ann Arbor, MI, 1991, also appeared in Prolog, Paris Papers 2, ISO/IEC JTC1 SC22 WG17 standardization report no. 80, 1991, pp. 87–130.

[17] E. Börger and D. Rosenzweig, The WAM — Definition and compiler correctness, in: C. Beierle and L. Plümer, eds., *Logic Programming: Formal Methods and Practical Applications*, North-Holland Series in Computer Science and Artificial Intelligence (North-Holland, Amsterdam, 1995) 21–90; preliminary version appeared as Technical Report TR-14/92, Dipartimento di Informatica, Università di Pisa, 1992.

[18] E. Börger and D. Rosenzweig, The mathematics of set predicates in Prolog, in: *Proc. Kurt Gödel Symposium '93* Lecture Notes in Computer Science, Vol. 713 (Springer, Berlin, 1993) 1–13; also appeared in Prolog, Copenhagen Papers 2, ISO/IEC JTC1 SC22 WG17 standardization report no. 106, 1993, pp. 33–42.

[19] E. Börger and R. Salamone, CLAM specification for provably correct compilation of CLP(R) programs, in: E. Börger, ed., *Specification and Validation Methods* (Oxford Univ. Press, Oxford, 1995) 97–130.

[20] E. Börger and P. Schmitt, A formal operational semantics for languages of type Prolog III, in: E. Börger, H. Kleine-Büning, M. Richter and W. Schönfeld, eds., *Computer Science Logic*, Lecture Notes in Computer Science, Vol. 533 (Springer, Berlin, 1991) 67–79.

[21] S.K. Debray and P. Mishra, Denotational and operational semantics for Prolog, in: *J. Logic Programming* **5** (1988) 61–91.

[22] A. de Bruin and E.P. de Vink, Continuation semantics for Prolog with cut, in: *Theory and Practice of Software Engineering*, Lecture Notes in Computer Science, Vol. 351 (Springer, Berlin, 1989) 178–192.

[23] A. de Bruin and E.P. de Vink, Retractions in comparing Prolog semantics, in: B. Rovan, ed., *Mathematical Foundations of Computer Science 1990*, Lecture Notes in Computer Science, Vol. 452 (Springer, Berlin, 1990) 189–186.

[24] B. Demoen, On the implementation of catch and throw in WAM. Department of Computer Science, University of Leuven, Manuscript, July 1989. See also Report CW 96.

[25] P. Deransart and G. Ferrand, An operational formal definition of Prolog: A specification method and its application, *New Generation Comput.* **10**(2) (1992) 121–171.

[26] P. Glavan and D. Rosenzweig, Communicating evolving algebras, in: E. Börger, G. Jäger, H. Kleine Büning, S. Martini and M.M. Richter, eds., *Computer Science Logic, Selected Papers from CSL '92*, Lecture Notes in Computer Science, Vol. 702 (Springer, Berlin, 1993).

[27] Y. Gurevich, Logic and the challenge of computer science, in: E. Börger, ed., *Trends in Theoretical Computer Science* (Computer Science Press, Rockville, MD, 1988) 1–57.

[28] Y. Gurevich, Evolving algebras. A tutorial introduction, *Bull. EATCS* **43** (1991) 264–284.
[29] C.A.R. Hoare, Preface to: D. Bjørner, C.A.R. Hoare and H. Langmaack, eds., *VDM '90. VDM and Z — Formal Methods in Software Development*, Lecture Notes in Computer Science, Vol. 428 (Springer, Berlin, 1990).
[30] N.D. Jones and A. Mycroft, Stepwise development of operational and denotational semantics for Prolog, in: *Proc. Internat. Symp. on Logic Programming 2/84, Atlantic City* (IEEE, 1984) 289–298.
[31] J.N. Kok, Specialization in logic programming: from Horn clause logic to Prolog and Concurrent Prolog, in: J.W. de Bakker, W.-P. de Roever, G. Rozenberg, eds., *Stepwise refinement of distributed systems. Models, formalisms, correctness*, Lecture Notes in Computer Science, Vol. 430 (Springer, Berlin, 1990), 401–413.
[32] T.G. Lindholm and R.A. O'Keefe, Efficient implementation of a defensible semantics for dynamic Prolog code, in: *Proc. 4th Internat. Conf. on Logic Programming* (Springer, Berlin, 1987) 21–39.
[33] J. Lloyd, *Foundations of Logic Programming* (Springer, Berlin, 2nd ed, 1987).
[34] J. Lloyd, Current theoretical issues in logic programming, Abstract, *Bull. EATCS* **39** (1989) 211.
[35] A. Martelli and G. Rossi, On the semantics of logic programing languages, in: *3rd Internat. Conf. on Logic Programming*, London, 1986, Lecture Notes in Computer Science, Vol. 225 (Springer, Berlin, 1986) 327–334.
[36] B. Müller, A semantics for hybrid object-oriented Prolog systems, in: B. Pehrson and I. Simon, eds., *Proc. 13th World Computer Congress '94, Vol. 1* (North-Holland, Amsterdam, 194) 428–433.
[37] T. Nicholson and N. Foo, A denotational semantics for Prolog, *ACM Trans. Programming Lang. Systems* **11** (1989) 650–665.
[38] N. North, A denotational definition of Prolog, National Physics Laboratory, Teddington, Report DITC 106/88.
[39] R. O'Keefe, A formal definition of Prolog, University of Auckland, BSI PS/22.
[40] G. Plotkin, A structural approach to operational semantics, Internal Report, CS Department, Aarhus University, DAIMI FN-19.
[41] *Quintus Prolog Reference Manual*, version 10, February, 1987. Quintus Computer Systems, Mountain View, CA.
[42] B.J. Ross and P.F. Wilkie, An algebraic semantics of sequential Prolog control, Department of AI, University of Edinburgh, DAI Research Paper 469.
[43] D. Scott, Outline of a mathematical theory of computation, Tech. Monograph PRG-2, November 1970, Oxford University Comp. Lab., Programming Res. Group, 1–24.
[44] D. Scott and C. Strachey, *Toward a Mathematical Semantics for Computer Languages*, Tech. Monograph PRG-6, August 1971, Oxford University Comp. Lab., Programming Res. Group, 1–42.
[45] D.H.D. Warren, An abstract Prolog instruction set, Tech. Note 309, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1983.
[46] M. Wirsing, Algebraic specification, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B* (Elsevier, Amsterdam, 1990) 675–788.
[47] PROLOG. Part 1, General Core, Committee Draft 1.0, ISO/IEC JTC1 SC22 WG17 N.92, 1992.