The WAM—Definition and Compiler Correctness*

Egon Börger
Dip. di Informatica
Universita di Pisa
Cso Italia 40
I-56100 PISA
boerger@di.unipi.it

Dean Rosenzweig
FSB
University of Zagreb
Salajeva 5
41000 Zagreb, Croatia
dean@math.hr

Abstract

This paper provides a mathematical analysis of the Warren Abstract Machine for executing Prolog and a correctness proof for a general compilation scheme of Prolog for the WAM. Starting from an abstract Prolog model which is close to the programmer's intuition, we derive the WAM methodically by stepwise refinement of Prolog models, proving correctness and completeness for each refinement step. Along the way we explicitely formulate, as proof assumptions, a set of natural conditions for a compiler to be correct, thus making our proof applicable to a whole class of compilers.

The proof method provides a rigorous mathematical framework for the study of Prolog compilation techniques. It can be applied in a natural way to extensions and variants of Prolog and related WAMs allowing for parallelism, constraint handling, types, functional components—in some cases it has in fact been successfully extended. Our exposition assumes only a general understanding of Prolog. We reach full mathematical rigour, without heavy methodological overhead, by using Gurevich's notion of evolving algebras.

*In: *Logic Programming: Formal Methods and Practical Applications*, C.Beierle,L.Plümer, Eds.), North–Holland, Series in Computer Science and Artificial Intelligence, 1995, pp. 20–90 (chapter 2).

Contents

Introduction

We provide a mathematical analysis of an implementation method—the Warren abstract machine for executing Prolog—and a proof of its correctness. The analysis has the flavour of a rational reconstruction, i.e. deriving the WAM methodically from a description of Prolog, and thus contributing, we hope, to understanding of its design principles. It proceeds by stepwise refinement of Prolog models, starting from an abstract, phenomenological, programmer's model. The proof follows the analysis closely, and consists in a chain of local proofs, of (relative) correctness and completeness for each refinement step.

The WAM, as described by [Warren 83], resembles an intricate puzzle, whose many pieces fit tightly together in a miraculous way. We analayze the complex web of their interdependencies, attempting to isolate the orthogonal components along the following lines.

As a first step, (operational semantics of) Prolog may be seen as decomposed into disjunctive, conjunctive and unificational components, which corresponds nicely to WAM handling of clause selection (predicate structure), continuations (clause structure) and representation of terms and substitutions.

We start therefore, in section 1, by defining an operational semantics of Prolog in which these components appear abstractly. The model comes in the form of *Prolog trees*, close to the usual intuitive picture and to its proof–theoretical logical background of SLD-trees. It is easily shown to be correct wrt SLD-resolution (for pure Prolog programs), and has been extended to a transparent yet rigorous formulation of the full language [Boerger,Rosenzweig 93], although we restrict it here to pure Prolog with *cut*. We thus start from scratch. In the same section we refine to a stack model (to the very same stack model on which the meaning of database operations was analyzed in [Boerger,Rosenzweig 91b], elaborating on the stack model of [Boerger 90a]). A stack model may be viewed as linear layout of Prolog tree traversal into 'computer memory' which brings us closer to the implementation view of the WAM.

Section 2 provides a complete account of WAM clause selection, i.e. representation of predicate structure. Since selection of clauses is largely independent of what clauses are, a 'compiler' model which emits WAM indexing and switching instructions, interspersed with abstract Prolog clauses, suffices for that purpose.

In section 3 we take up the continuations, i.e. clause structure. Since Prolog terms and substitutions are still kept abstract, i.e. unrepresented, whereas full WAM treatment of continuations does partially depend on term representation, we arrive, on this level, to a simplified 'compiler' which encodes a clause

$$H :- G_1, \ldots, G_n$$

with a (pseudo)instruction sequence

$$\mathit{allocate}, \mathit{unify}(H), \mathit{call}(G_1), \ldots, \mathit{call}(G_n), \mathit{deallocate}, \mathit{proceed}.$$

In section 4 we analyze the representation of terms and substitutions. Correctness and completeness are first proved for a simplified model, in which all variables are permanent, getting initialized to *unbound* as soon as they are allocated. The fine points of the WAM—environment trimming, local and unsafe values, last call optimization, Warren's classification of variables and their on-the-fly initialization—are then introduced as local optimizations and related correctness preserving devices. The notion of a variable being *needed* at a body goal turns out to be the crucial hidden concept which clarifies them all.

We thus arrive at the full WAM of [Warren 83], and at a proof of its correctness and completeness wrt Prolog trees (Main Theorem in 4.3). There was no need to prescribe a specific compiler—the cumulative assumptions, that we explicitly introduce along the way, provide a set of sufficient conditions for a compiler to be correct. The assumptions might be most useful in the role of a clear prescription for the compiler writer, of what he must do to create a correct compiler. They can be useful in very much the same way when trying to extend and/or modify the WAM. The proof provides a clear scheme to such designers for proving *their* WAM to be correct, without need for extensive testing of all components, including the well known ones.

We feel that the mathematical beauty of the WAM gets revealed in the process. We join [Ait-Kaci 91] in saying, from a different context, that

> ...this understanding can make one appreciate all the more David H.D.Warren's feat, as he conceived this architecture in his mind as one whole.

The approach would have been impossible without a methodology allowing for stepwise refinement of abstraction level, yet retaining full mathematical rigour at all levels, without however imposing heavy methodological overhead. It is the straightjacket of fixed abstraction level, and the induced combinatorial explosion of formalism and/or mathematics, which usually makes semantical investigation of real situations so difficult.

The method of *evolving algebras* of [Gurevich 88,91], see also [Glavan,Rosenzweig 93], that we have adopted (here and in [Boerger,Rosenzweig 91a-d,93], continuing the work in [Boerger 90a,90b]), allows precisely the kind of fine-tuning of abstraction level to the problem at hand, and natural modelling of dynamics at all levels, that is required here. Evolving algebra descriptions may be easily read as 'pseudocode over abstract data'. We indeed do encourage the reader, who is (maybe on first reading) primarily interested in analysis rather than proof, to read the paper precisely from such an intuitive viewpoint, without worrying much about evolving algebras (skipping to the next subsection as soon as we start talking about proof or 'proof-maps'). A glance at Evolving Algebras section should suffice for such a reading.

The reader who is interested in checking the proof should study that section more carefully. Appendici 1–4 might assist a proof checking reader by summarizing the transition rule systems of some Prolog models constructed at different abstraction levels. It may need stressing that we provide a proof in the usual sense of (informal) mathematics, and not a derivation in any formal deductive system. Our attitude here is preformal rather than antiformal—useful deductive systems for evolving algebras have yet to be developed.

All we assume of the reader is general understanding of Prolog as a programming language. Although the paper is technically self–contained, familiarity with the WAM might help. For extended discussion of basic concepts of the WAM we refer the reader to [Ait-Kaci 91]. For more background on evolving algebra methodology the reader is referred to [Gurevich 91].

There have been other attempts at analysis and correctness proof of the WAM in the literature. The book [Ait-Kaci 91] provides a penetrating but informal tutorial analysis, without any pretension to proof, proceeding by elaborating successive partial Prologs at the same level of abstraction (unlike our successive partial elaborations of full Prolog). [Russinoff 92] proves correctness of a partial (simplified) WAM wrt a form of SLD-resolution, relying on *ad hoc* arithmetical encoding and a specific compiler. His formidable proof can hardly be understood as conceptual analysis, and does not seem to be easily extendible either to richer (in built-in predicates and constructs) or to similar languages. We feel that a convincing analysis and a proof should go together, i.e. be essentially the same thing, and that similarity of 'similar languages' should be explicitly captured.

The paper is a synthesis and a revision of [Boerger,Rosenzweig 91a,c], using [Boerger,Rosenzweig 93]. It simplified a lot to deal with the or–structure before the and–structure, and to base the or–structure treatment on trees instead of stacks. Semantics and implementation of dynamic code and database operations have been analyzed in [Boerger,Rosenzweig 91b], by extending some of the models used here. The work of [Beierle,Boerger 92] on Protos-L, a Prolog with types, and [Börger,Salamone 94] on constraint propagation language CLP(R), where the construction and the proofs were smoothly ported to the 'PAM' [Beierle,Boerger 92] and the 'CLAM'

[Börger,Salamone 94], confirms the conjecture from [Boerger,Rosenzweig 91a], that '…our explication should essentially apply to WAM-based implementation of other logic languages with the same and-or structure as Prolog.'

Acknowledgements. We are indebted to Yuri Gurevich for permanent discussion on the whole evolving algebra framework. Many people have helped us with useful comments on various early versions of this work—we thank them all, mentioning here just, with apologies to omissions, Chris Beierle, Michael Hanus, Davorin Ruševljan, Jörg Süggel and the Ph.D. students of Dipartimento di Informatica in Pisa 1992—Agostino Dovier, Stefano Guerrini, Angelo Monti, Corrado Priami, Paola Quaglia, Alessandro Roncato, Luca Roversi—who have, after two weeks of initiation to the framework and to the WAM, cheerfully checked all the proofs, caught some slips and raised several good points. Special thanks go to Jörg Süggel also for very careful reading of the final version and valuable detailed comments.

Evolving Algebras

All Prolog models, constructed in this paper by stepwise refinement, are *evolving algebras*, which is a notion introduced by [Gurevich 91]. 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

$$\textbf{if } R? \textbf{ then } R!$$

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' [Wirsing 90], as implying that both arguments are defined (see also [Glavan,Rosenzweig 93]).

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, for discussion cf. [Glavan,Rosenzweig 93]. 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

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

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,

$$\begin{array}{l} \textbf{if } \ldots a =< X, Y > \ldots \\ \textbf{then } \ldots X \ldots Y \ldots \end{array}$$

abbreviates

$$\begin{array}{l} \textbf{if } \ldots \mathrm{ispair}(a) \ldots \\ \textbf{then } \ldots \mathrm{fst}(a) \ldots \mathrm{snd}(a) \ldots, \end{array}$$

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:

$$\textbf{extend } A \textbf{ by } t_1, \ldots, t_n \textbf{ with } \textit{updates} \textbf{ endextend}$$

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 91] 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 [Glavan,Rosenzweig 93]).

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.

As the set of updates executed in a rule is to be understood as executed simultaneously, we shall at several points explicitly indicate that *sequential* execution of updates is intended, by an update form

$$\textbf{seq } i = 1, \ldots, n \ \mathit{update}(i) \textbf{ endseq}$$

where n is a numerical term and $\mathit{update}(i)$ is an update (with parameter i). We intend an update sequence to trigger sequential execution of $\mathit{update}(i)$ (for $i = 1, \ldots, n$) so that the overall system does not attempt another rule application before $\mathit{update}(n)$ has been executed. This construct, which does not appear in Gurevich's original definition in [Gurevich 91] is reducible to rules with function updates, as the reader might verify as an exercise.

In our refinement steps we shall typically construct a 'more concrete' evolving algebra B and relate it to a 'more abstract' evolving algebra A by a (partial) *proof map* $\mathcal{F}$ mapping states $\mathcal{B}$ of B to states $\mathcal{F}(\mathcal{B})$ of A, and rule sequences R of B to rule sequences $\mathcal{F}(R)$ of A, so that the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "\mathcal{F}(R)"] & A' \\
B \arrow[u, "\mathcal{F}"] \arrow[r, "R"'] & B' \arrow[u, "\mathcal{F}"']
\end{tikzcd}$$

We shall consider such a proof map to establish *correctness* of $(\mathcal{B}, \mathcal{S})$ with respect to $(\mathcal{A}, \mathcal{R})$ if $\mathcal{F}$ preserves initiality, success and failure (value of *stop*) of states, since in that case we may view successful (failing) concrete computations as implementing successful (failing) abstract computations.

We shall consider such a proof map to establish *completeness* of $(\mathcal{B}, \mathcal{S})$ with respect to $(\mathcal{A}, \mathcal{R})$ if every terminating computation in $(\mathcal{A}, \mathcal{R})$ is image under $\mathcal{F}$ of a terminating computation in $(\mathcal{B}, \mathcal{S})$, since in that case we may view every successful (failing) abstract computation as implemented by a successful (failing) concrete computation.

In case we establish both correctness and completeness in the above sense, as we do on every of our refinement steps, we may speak of *operational equivalence* of evolving algebras (in this restricted context, where only deterministic terminating computations are of interest). Since this last notion is symmetric, it does not matter any more which way $\mathcal{F}$ goes. The reader will notice that we indeed have, in a few places, found it more convenient to reverse the direction of $\mathcal{F}$, mapping the ' abstract' algebra into the 'concrete' one.

1 Prolog—Tree and Stack Models

In this section we define Prolog trees, the abstract Prolog model with respect to which we will prove the WAM to be correct. This mathematical model captures the usual intuitive operational understanding of Prolog, what justifies its use as basis for our correctness proof. It is also close to SLD-resolution proof trees, (and allows a simple correctness proof with respect to the latter, for pure Prolog programs, [Boerger,Rosenzweig 93]). We then show how to lay out these trees linearly, thus transforming the tree into a stack model which is closer to what is needed for a mathematical analysis of the stack based WAM.

1.1 Prolog Tree

A Prolog computation can be seen as systematic search of a space of possible solutions to initially given query. The set of computation states (*resolution states* or *configurations* or *instantaneous descriptions*) is often viewed as carrying a tree structure, with 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 resolvent states* still to be tried, as we will explain below.

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

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

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*. 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 (in *Call mode*). The mode then turns to *Select*, and the first unifying son from $cands(n)$ gets visited (i.e. 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*.

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

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

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 goals still to be executed. In view of the *cut* operator *!*, however, this sequence will not be represented linearly, but structured into subsequences—clause bodies decorated with appropriate *cutpoints*, i.e. backtracking states current when the clause was called. We therefore introduce a universe and a function

$$\begin{aligned} DECGOAL &= GOAL \times NODE \\ decglseq &: NODE \rightarrow DECGOAL^* \end{aligned}$$

associating the relevant sequence of (decorated) goals to each node.

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 functions

$$\begin{aligned} unify &: TERM \times TERM \rightarrow SUBST \cup \{nil\} \\ subres &: DECGOAL^* \times SUBST \rightarrow DECGOAL^* \end{aligned}$$

where *unify* is an abstract unification function associating to two terms either their unifying substitution or the answer that there is none [1]; *subres* is a substitution applying function yielding the result of applying the given substitution to all goals in the sequence; we further assume the substitution concatenating function $\circ$.

We represent renaming of variables abstractly, without going into details of term and variable representation, by introducing a function

$$rename : TERM \times \mathcal{N} \rightarrow TERM$$

renaming all variables in the given term at the given index (renaming level). The current renaming index—the one to be used for the next renaming—is indicated by a 0-ary function *vi*.

The above mentioned switching of *modes* will be represented using 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 $stop \in \{0, 1, -1\}$, to indicate respectively running of the system, halting with success and final failure. In a similar manner we could (but shall not in this paper) handle error conditions by a distinguished element *error*, taking values in a set of error messages. Besides this we will use (and consider as part of Prolog tree algebras) all the usual *list operations* for which we adopt standard notation. In the same way we shall use *hd* and *bdy* to select heads and bodies of clauses, allowing ourselves the freedom to confuse a list of literals with their iterated conjunction, suppressing the obvious translation. Codomain of *bdy* will thus be taken to be $TERM^*$.

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*).

The reader 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.

[1] We don't have, at this point, to be more explicit about our notion of unification—for more careful discussion see 4.2 and 4.3

Wishing 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

$$\begin{aligned} clause &: CODE \rightarrow CLAUSE \\ cll &: NODE \rightarrow CODE \end{aligned}$$

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

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

of which we assume to yield the (properly ordered) list of the candidate clause occurrences for the given literal in the given program. The current program is represented by a distinguished element *db* of *PROGRAM* (the *database*). Note that existence of *procdef* is all that we assume of the abstract universe *PROGRAM*.

This concludes the definition of the signature of Prolog tree algebras. Notationally, we usually suppress the parameter *currnode* by writing simply

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

Components of 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*.

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 to reach a state with $stop = 1$ (due to first successful execution of the query) or with $stop = -1$ (due to its final failure by backtracking all the way to *root*). We introduce the following abbreviation for backtracking to father:

```
backtrack ≡ if $father = root$
            then $stop := -1$
            else $currnode := father$
                 $mode := Select$
```

We take the *stop* value of 1 or -1 to represent abstractly the final success or failure of the query, outputting the substitution or providing a 'no (more) solutions' answer respectively. No transition rule will be applicable in such a case, which is a natural notion of 'terminating state'. All transition rules will thus be tacitly assumed to stand under the guard

$$\text{OK} \quad \equiv \quad stop = 0$$

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

```
if $all\_done$
then $stop := 1$
```

where *all_done* abbreviates $decglseq = [\,]$. [2] 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 $decglseq := rest(decglseq)$
```

The existence of *goal*, assumed in the guard, is understood as excluding *all_done*, cf. above abbreviations. Likewise, the existence of *act*, assumed in rules to follow, is understood as excluding both *all_done* and $goal = [\,]$ conditions—we shall, in general, tacitly understand guards, relying on existence of some objects, as excluding all conditions under which these objects could be undefined, suppressing the obvious boolean conditions which would formally ensure such exclusion. In *goal success* rule e.g. the suppressed conditions is $NOT(all\_done)$, and in rules mentioning *act* below it is $NOT(all\_done)$ & $goal \neq [\,]$.

The crucial resolution step, applicable to a user defined predicate, 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
  extend $NODE$ by $temp_1, \ldots, temp_n$ with
    $father(temp_i) := currnode$
    $cll(temp_i) := nth(procdef(act, db), i)$
    $cands := [temp_1, \ldots, temp_n]$
  endextend
  $mode := Select$
  where $n = length(procdef(act, db))$
```

where *is_user_defined* is a boolean function recognizing those literals whose predicate symbols are user defined (as opposed to built-in predicates and language constructs). Note that goals and substitutions, attached to candidate sons, are at this point undefined, and that the value of *currnode* does not change. [3]

The following **selection rule**, applicable to (visited) nodes with user defined activator in *Select* mode, attempts to select a candidate resolvent state (selecting thereby the associated clause occurrence). If there is none, the system backtracks. If the renamed head of selected clause does

[2] If the reader wants to view Prolog as returning all solutions, all he has to do is to modify this rule so as to trigger backtracking.

[3] The expert will notice that given database operations of full Prolog this rule would commit us to the so-called logical view.

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, getting thus visited, gets erased from its father's *cands* list), its 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. A technicality: since the current variable renaming index *vi* has now been used, it should be updated to a fresh value for subsequent use.

```
if $is\_user\_defined(act)$
   $\&\ mode = Select$
thenif $cands = [\,]$
then $backtrack$
elsif $unify = nil$
then $cands := rest(cands)$
else $currnode := fst(cands)$
       $decglseq(fst(cands)) :=$
         $subres([\,\langle bdy(clause), father\rangle \mid cont\,], unify)$
       $s(fst(cands)) := s \circ unify$
       $cands := rest(cands)$
       $mode := Call$
       $vi := vi + 1$
where  $clause = rename(clause(cll(fst(cands))), vi)$
       $unify = unify(act, hd(clause))$
```

The rules for *true, fail* are self explaining:

```
if $act = true$          if $act = fail$
then $succeed$           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 $father := cutpt$
     $succeed$
```

This concludes the list of rules for our model of 'pure Prolog with *cut*'. The model could be extended to cover various built-in predicates and language constructs simply by adding appropriate rules, like in [Boerger,Rosenzweig 93]. The reader might have noticed that the guards for rules *goal success, cut, true, fail* 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*. It is also easy to see that 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.

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

- **visited** 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*;

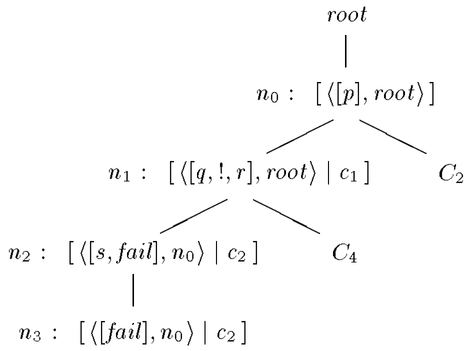
- **abandoned** if it is visited but not active, i.e. has been backtracked from;
- **candidate** if it belongs to *cands* list of an active node.

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

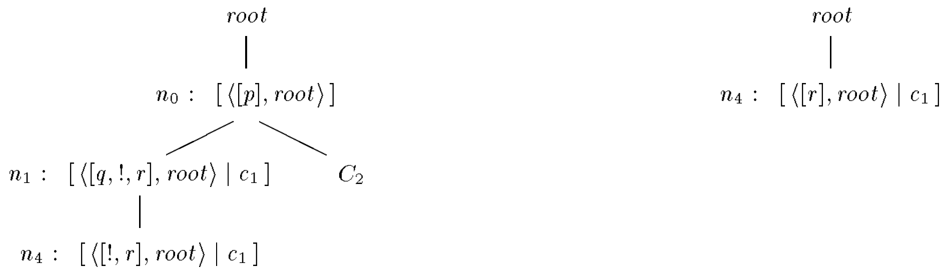
Example 1. Given the Prolog program

$$\begin{array}{lll} p & :- & q, !, r. \\ p. & & \\ q & :- & s, \mathit{fail}. \\ q. & r. & s. \end{array}$$

and query p, with clauselines $C_i,\ i = 1, \ldots, 6$, associated in order, we would, in a few steps, arrive at the following Prolog tree.


$root$
n_0 : $[\langle [p], root\rangle]$
n_1 : $[\langle [q, !, r], root\rangle \mid c_1]$
C_2
n_2 : $[\langle [s, \mathit{fail}], n_0\rangle \mid c_2]$
C_4
n_3 : $[\langle [\mathit{fail}], n_0\rangle \mid c_2]$

Only the active nodes are shown in the picture—for visited nodes we display a label and *decglseq*, while candidate nodes are displayed just with *cll*. The reader should be able to reconstruct the continuations c_1, c_2 himself. Executing further *fail*, then backtracking (twice), *call, selection* and *query success*, we arrive to execution of *cut*, for which we show the tree (of active nodes) *before* and *after*:


$root$
n_0 : $[\langle [p], root\rangle]$
n_1 : $[\langle [q, !, r], root\rangle \mid c_1]$
C_2
n_4 : $[\langle [!, r], root\rangle \mid c_1]$
$root$
n_4 : $[\langle [r], root\rangle \mid c_1]$

Executing another *call, selection* and thrice *goal success* would finally invoke *query success.*

The classification of nodes suggests connections both to SLD-tree [Apt 90, Lloyd 84] and to stack models of Prolog, to be elaborated in subsequent sections. In [Boerger,Rosenzweig 93] its correctness wrt SLD-resolution was established.

1.2 From Tree to Stack

Classification of Prolog tree nodes suggests a straightforward *stack* representation: disregarding the abandoned nodes (since they, once abandoned, play no further role in the computation), we may view the path of active nodes as a stack—if *cands* lists are represented elsewhere.

We already have the *CODE* universe, which, given some additional structure, may easily represent sequencing of clauses in a Prolog program: it will be provided with a *successor* function +, modelling the linear structure of all *cands* lists, and codomain of *clause* will obtain a special *nil* value to indicate end of list. Thus, *CODE* gets refined to, say, *CODEAREA*, with

$$\begin{array}{rcl} + & : & CODEAREA \rightarrow CODEAREA \\ cll & : & NODE \rightarrow CODEAREA \\ clause & : & CODEAREA \rightarrow CLAUSE + \{nil\} \\ procdef & : & LIT \times PROGRAM \rightarrow CODEAREA \end{array}$$

Note that *procdef* now yields an element of *CODEAREA*, i.e. a pointer, instead of a list; the old list can easily be reconstructed using a function $cll s : CODEAREA \rightarrow CODEAREA^*$ such that

$$\begin{array}{rcl} clls(Ptr) & = & \textbf{if } clause(Ptr) = nil \\ & & \textbf{then } [\,] \\ & & \textbf{else } [\,Ptr \mid clls(Ptr+)\,] \end{array}$$

assuming now that $clls(procdef(G, Db))$ yields the same as $procdef(G, Db)$ of previous section, i.e. the list of (pointers to) all candidate clause(occurrence)s, in proper ordering.

We shall further separate the information contained in *currnode* ¿from other active nodes, its *backtracking points* or *choicepoints*, by recording the former in 0-ary functions ('registers' of an abstract machine)

$$decglseq \qquad s \qquad cll$$

typed like homonymous abbreviations of previous sections. It may then be (stylistically) appropriate to rename *NODE* to *STATE*, *root* to *bottom*, *father*(*currnode*) and *father* to 0−ary and unary *b* (for backtracking),

$$b \in STATE \qquad b : STATE \rightarrow STATE$$

replacing our tree algebra

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

by three 'registers' above and *stack algebra*

$$(STATE; bottom, b; b)$$

More formally, we will have a mapping $\mathcal{F}$ which maps stack elements to tree nodes as:

$$(decglseq, s, cll, b, bottom, vi) \longrightarrow (currnode, root, vi)$$

where the node decorations in the tree are recovered from the registers and the decorations of stack elements as follows:

$$\begin{array}{rcl} decglseq(currnode) & = & decglseq \\ s(currnode) & = & s \\ cands(currnode) & = & mk\_cands(currnode, cll) \\ father(currnode) & = & F(b) \end{array}$$

with $F : \mathit{STATE} \rightarrow \mathit{NODE}$ an auxiliary function such that

$$\begin{aligned}
\mathit{decglseq}(F(n)) &= \mathit{decglseq}(n) \\
s(F(n)) &= s(n) \\
\mathit{cands}(F(n)) &= \mathit{mk\_cands}(F(n), \mathit{cll}(n)) \\
\mathit{father}(F(n)) &= F(b(n)) \\
F(\mathit{bottom}) &= \mathit{root}
\end{aligned}$$

where

```
$mk\_cands(Node, Cll)$  ≡  if $clause(Cll) = nil$
                           then $[\,]$
                           else $[\,\langle Node, Cll\rangle \mid mk\_cands(Node, Cll+)\,]$,
```

keeping in mind that candidate nodes of the previous section can be thought of as $\langle \mathit{father}, \mathit{cll}\rangle$ pairs, since these are their only decorations.

Rules *goal* and *query success, true, fail, cut* of the previous section may be retained in the present setup (translated to our stack algebra signature); the *backtrack* update will now take the form

```
$backtrack$  ≡  if $b = bottom$
                then $stop := -1$
                else $fetch\_state\_from(b)$
                     $b := b(b)$
                     $cll := cll(b)$
                     $mode := Select$
```

where

```
$fetch\_state\_from(l)$  ≡  $decglseq := decglseq(l)$
                            $s := s(l)$
```

The *backtrack* update may then be thought of as 'popping' the stack, unloading the 'contents' of its top to 'registers'. [4]

Rules *call* and *selection* should be slightly modified to allow for new data representation. *Call* rule will now, instead of creating candidate sons, simply set *cll*, taking the form

```
if $is\_user\_defined(act)$
   & $mode = Call$
then $cll := procdef(act, db)$
     $mode := Select$
```

[4] The updates of *cll* and *b* are not included in *fetch_state_from* update just in order to allow the latter to survive till compilation of predicate structure in 2.2.

Selection rule will now do the creating, *pushing* a single new choicepoint:

```
if $is\_user\_defined(act)$
   & $mode = Select$
thenif $clause(cll) = nil$
then $backtrack$
elsif $unify = nil$
then $cll := cll+$
else push $temp$ with
        $store\_state\_in(temp)$
        $cll(temp) := cll+$
     endpush
     $decglseq :=$
        $subres([\langle bdy(clause), b\rangle \mid cont], unify)$
     $s := s \circ unify$
     $mode := Call$
     $vi := vi + 1$
where $clause = rename(clause(cll), vi)$
      $unify = unify(act, hd(clause))$
```

where we use the mnemonic abbreviations:

```
push $t$ with        $\equiv$   extend $STATE$ by $t$ with
  $updates(t)$                $b := t$
endpush                       $b(t) := b$
                              $updates(t)$
                            endextend

$store\_state\_in(t)$  $=$   $decglseq(t) := decglseq$
                            $s(t) := s$
```

The reader might care to simulate Example 1 on the current stack model.

Note that our 'stack' still carries a tree structure: when we 'pop' it on backtracking, we don't discard the popped states in any 'physical' sense; they are still there[5] and may be used when needed. We need them now to establish a complete correspondence to the tree of previous section, using F. The structure of visited nodes is then completely preserved. Defining further $\mathcal{F}$ on rules as homonymy, it is now straightforward to verify that rule execution commutes with $\mathcal{F}$, yielding

Proposition 1.2. The stack model of 1.2. is correct and complete wrt Prolog trees.

Remark. As explained in detail in the section on evolving algebras, all our correctness proofs are of this form: a map $\mathcal{F}$ will be defined which maps refined data structures into the more abstract data structures, and (sequences of) refined rules into (sequences of) more abstract rules in such a way that the execution of these rule sequences commutes with $\mathcal{F}$. Once the right functor $\mathcal{F}$ is found, the commutativity of the corresponding diagram is usually easy to prove.

1.3 Reusing Choicepoints

A natural step of 'optimizing' our model will direct us firmly towards the WAM representation of disjunctive structure of Prolog.

[5] Talking about a 'stack' here is then just a shift of emphasis. This situation will change only in section 4.

The stack model, derived in the previous section by a straightforward representation of Prolog trees, is, when viewed as a model for implementation, not very realistic. It namely separates the unifiability test from actual update of computation state, pushing a choicepoint, in selection rule, only on successful unification. This choicepoint is later, if the selected unifying clause fails, popped, just to push another, very similar (differing just in the value of *cll*) choicepoint for the next unifying clause etc.

In an implementation, however, an attempt at unification involves massive side effects, modifying the whole computation state. Old state, to be restored on backtracking, should then be stored *somewhere, before* any test of unifiability. A choicepoint had then better be pushed *before* unification (if it is to be pushed at all, cf. discussion of *determinacy detection* in the next section), and simply reused for all alternative clauses for the same call, by updating its *cll* value.

We shall then, retaining essentially the signature ('data types') of the previous section, decompose the action of selection rule into more primitive steps, attempting to reorganize them in a more 'efficient' way. The sequencing of these steps will be controlled by 0-ary function *mode*, which will now take more values, by decomposing old *Select* mode.

Pushing a choicepoint is, at this level of abstraction, one primitive action, invoked by *mode* value of *Try*. Attempting unification and updating *decglseq, s* on success is our second primitive, invoked by *mode* value of *Enter*. Reusing the choicepoint by invoking an alternative clause is the third primitive, invoked by *Retry* mode. Old *Call* mode will retain its role, taking over the first clause of *Select* mode as well, i.e. triggering immediate backtracking in case of no clauses.

Minding that *Enter* mode should update *decglseq* with the *old* value of b as cutpoint, while *Try*, to be executed before *Enter*, modifies b by pushing, we introduce a new 0-ary function ('cutpoint register') $ct \in STATE$ which will, in call mode, store b's old value for *Enter* to find.

The rules, then.

```
if $is\_user\_defined(act)$
    & $mode = Call$
thenif $clause(procdef(act, db)) = nil$
then $backtrack$
else $cll := procdef(act, db)$
     $mode := Try$
     $ct := b$

if $mode = Try$
then push $temp$ with
         $store\_state\_in(temp)$
         $cll(temp) := cll+$
     endpush
     $mode := Enter$
```

```
if $mode = Enter$
thenif $unify = nil$
then $backtrack$
else
  $decglseq :=$
      $subres(\ [\langle\ bdy(clause), ct\ \rangle \mid cont],$
              $unify\ )$
  $s := s \circ unify$
  $mode := Call$
  $vi := vi + 1$
where $clause = rename(clause(cll), vi)$
      $unify = unify(act, hd(clause))$
```

We shall have to distinguish two forms of backtracking now—reusing the choicepoint by activating the stored alternative clause (we shall call such an update *backtrack*), and popping it, when there is no alternative clause (the update will be called *deep_backtrack*).

```
$backtrack$  $\equiv$  if $b = bottom$
                  then $stop := -1$
                  else $mode := Retry$

$deep\_backtrack$  $\equiv$  if $b(b) = bottom$
                        then $stop := -1$
                        else $b := b(b)$
```

The choicepoint gets thus popped by *deep_backtrack*; reusing it, with 'unloading' of its contents, will be effected in *Retry* mode.

```
if $mode = Retry$
thenif $clause(cll(b)) = nil$
then $deep\_backtrack$
else $fetch\_state\_from(b)$
     $cll := cll(b)$
     $cll(b) := cll(b)+$
     $ct := b(b)$
     $mode := Enter$
```

Note that, since a choicepoint is pushed for every call (by *Try*), the correct cutpoint is always at $b(b)$. It is namely the value b held when the current choicepoint was pushed.

The reader might care to see, simulating Example 1 on the current model, how a choicepoint gets reused.

For sake of proof of correctness of this refined stack model, we shall distinguish between different actions formulated within a single rule. We shall accordingly talk about $Retry_1$ referring to *deep_backtrack* update of the first clause of *Retry* rule, and about $Retry_2$ referring to the composite update of the second clause. Likewise, we shall use $Enter_1$, $Enter_2$, $Call_1$, $Call_2$, and, referring to the previous section, $Selection_1$, $Selection_2$, $Selection_3$.

The state map $\mathcal{F}$ will be defined on all states in *Call* and *Retry* modes. This will obviously suffice, since, by inspection of the rules, after a *Call* another *Call* or *Retry* must come in at most three steps (after *Try* followed by *Enter*), while after a *Retry* another *Retry* or *Call* comes in at most two steps (after *Enter*). Note also that all initial and terminal states are included—final failure can occur only in *Call* or *Retry* mode. The map will however be differently defined for the two modes: for *Call* mode $\mathcal{F}$ is simply the identity, while for *Retry* mode $\mathcal{F}(St)$ will be obtained by popping the top of stack into 'registers'—more formally

$$\langle\, decglseq, s, cll, b\,\rangle(\mathcal{F}(St)) \quad = \quad \langle\, decglseq(b(St)), s(b(St)), cll(b(St)), b(b(St))\,\rangle,$$

retaining *vi* and setting *mode* to *Select*.

The rule map will, in addition to being homonymous on *query* and *goal success, true, fail, cut*, be defined by

$$\begin{aligned}
\mathcal{F}([\,Call_1\,]) &= [\,Call, Selection_1\,]\\
\mathcal{F}([\,Call_2, Try, Enter_1\,]) &= [\,Call, Selection_2]\\
\mathcal{F}([\,Call_2, Try, Enter_2\,]) &= [\,Call, Selection_3\,]\\
\mathcal{F}([\,Retry_1\,]) &= [\,Selection_1\,]\\
\mathcal{F}([\,Retry_2, Enter_1\,]) &= [\,Selection_2\,]\\
\mathcal{F}([\,Retry_2, Enter_2\,]) &= [\,Selection_3\,]
\end{aligned}$$

It is now straightforward to verify that $\mathcal{F}$ commutes with execution of respective rule sequences, proving

Proposition 1.3. The Prolog stack model of 1.3. is correct and complete wrt that of 1.2.

Here we have decomposed the action of *Select* rule into four more primitive actions, which will pave the way for separate refinement, in the sequel, of predicate structure—as represented here by *Try, Retry* modes, from that of clause structure—as represented by *Call, Enter* modes.

2 Predicate Structure

In this section we analyse the way the disjunctive structure of predicates (the selection of clauses) in a Prolog program is to be compiled for the WAM. We introduce switching instructions but with an abstract indexing scheme.The structure of Prolog clauses and terms is not relevant for this analysis and is therefore kept abstract here.

2.1 Determinacy Detection (look-ahead optimization)

Pushing (by *Try* rule of previous section) a choicepoint St with $clause(cll(St)) = nil$, or updating (by *Retry*) its cll in such a way, has obviously the effect of creating a useless, silly choicepoint: were it ever *Retri*ed, it would be immediately popped by *deep_backtracking*. A simple modification would avoid pushing silly choicepoints at all:

Rules *Try* and *Retry* obtain additional look-ahead guards,

$$clause(cll{+}) \neq nil \qquad clause(cll(b){+}) \neq nil$$

respectively, preventing them from making silly choicepoints and making choicepoints silly. The modified rules are then

```
if $mode = Try$
then $mode := Enter$
     if $clause(cll+) \neq nil$
     then
        push $temp$ with
          $store\_state\_in(temp)$
          $cll(temp) := cll+$
        endpush
```

```
if $mode = Retry$
then $fetch\_state\_from(b)$
     $cll := cll(b)$
     $cll(b) := cll(b)+$
     $ct := b(b)$
     $mode := Enter$
     if $clause(cll(b)+) = nil$
     then $b := b(b)$
```

Note that, with the look-ahead, a *nil* clause will never get to $cll(b)$, so that the old guard $clause(cll(b)) \neq nil$ could be dropped, together with the *deep_backtrack* update.

Since recognizing the (potentially) current choicepoint as becoming silly amounts to detecting the current call to be determinate (i.e. to have no (more) alternative clauses), the look-ahead optimization is sometimes called 'determinacy detection', eg. [Lindholm,O'Keefe 87].

The reader might care to see, by simulating Example 1 on the current model, how creation of silly choicepoints is prevented.

The proof map $\mathcal{F}$, mapping now the algebra of 1.3. to that of 2.1, is rather obvious: the rule map is identity, while the state map skips all silly choicepoints, those with $clause(cll) = nil$, recursively down the stack. More formally, we can use functions $F : STATE \rightarrow STATE$ and $G : DECGOAL^* \rightarrow DECGOAL^*$ such that

$$\begin{array}{rcl} F(St) & = & \textbf{if } St = bottom \textbf{ then } bottom \\ & & \textbf{elsif } clause(cll(St)) = nil \textbf{ then } F(b(St)) \\ & & \textbf{else } \langle\, G(decglseq(St)), s(St), cll(St), F(b(St)) \,\rangle \end{array}$$

$$\begin{array}{rcl} G([\,]) & = & [\,] \\ G([\langle goal, cutpt\rangle \mid rest]) & = & \textbf{if} cutpt = bottom\textbf{then}[\langle goal, cutpt\rangle \mid G(rest)] \\ & & \textbf{elsif} clause(cll(cutpt)) = nil\textbf{then}G([\langle goal, b(cutpt)\rangle \mid rest]) \\ & & \textbf{else}[\langle goal, cutpt\rangle \mid G(rest)] \end{array}$$

setting then

$$\langle\, decglseq, s, cll, b\rangle(\mathcal{F}(St)) \quad = \quad \langle\, decglseq(St), s(St), cll(St), F(b(St)) \,\rangle,$$

retaining *vi* and *mode*. Straightforward verification now yields

Proposition 2.1. Determinacy detection preserves correctness, i.e. the model of 2.1. is correct and complete wrt that of 1.3.

2.2 Compilation of Predicate Structure

We are now ready for a detailed analysis of the way the disjunctive structure of Prolog predicates is to be compiled into code for the WAM. A predicate will be represented as (a pointer to) a sequence of dedicated instructions which will determine creation, reuse and discarding of choicepoints. Since it is largely independent of the exact way how the structure of Prolog clauses and terms is compiled, we may, for the time being, abstract the latter away, assuming that *CODEAREA* stores indexing instructions [Warren 83, Ait-Kaci 91] interspersed with proper clauses. The resulting model may be seen as elaborating *only* the disjunctive, predicate structure of a Prolog program, providing thus a splendid example of freedom, inherent to the evolving algebra approach, of fine-tuning the level of abstraction to the problem at hand.

We thus have to modify slightly our signature. Stylistically, we replace *cll,clause* with *p* (for 'program pointer') and *code*, assuming temporarily, for technical reasons, a special 'location' *start*. We thus have

$$\begin{array}{rcl} p, start & \in & CODEAREA \\ code & : & CODEAREA \;\rightarrow\; INSTR + CLAUSE + \{nil\} + \{code(start)\} \end{array}$$

where

$$\begin{array}{rcl} INSTR & = & \{\; try\_me\_else(N), retry\_me\_else(N), trust\_me(N), \\ & & \;\; try(N), retry(N), trust(N) \mid N \in CODEAREA \,\} \end{array}$$

The *INSTR* universe will be tacitly enlarged in the sequel, as we introduce more WAM instructions.

Changing from *clause*, *cll* to *code*, *p* may be seen as largely cosmetic. What is however important is that the operation of our old algebras in *Try, Retry* modes will now be simulated by *executing instructions*, *try_me_else* or *try* in case of *mode* = *Try*, *retry_me_else*, *retry*, *trust_me*, *trust* in case of *mode* = *Retry*. Execution will be controlled by setting the program pointer *p*. The modes may be entirely dispensed with, using $p = start$, $code(p) \in CLAUSE$ as replacements for *Call, Enter* modes, respectively.

The backtracking updates will hence reset the program pointer, instead of the mode:

$$backtrack \;\equiv\; \begin{array}{l} \textbf{if } b = bottom \\ \textbf{then } stop := -1 \\ \textbf{else } p := p(b) \end{array}$$

Of *procdef*(*G*, *Db*) we shall now assume to return a pointer either to a single clause or to a program section looking like

$$\begin{array}{rcccrc} N_1 : & try\_me\_else(N2) & & \text{or} & N_1 : & try(C_1) \\ & \vdots & & & & \vdots \\ N_2 : & retry\_me\_else(N3) & & & N_2 : & retry(C_2) \\ & \vdots & & & & \vdots \\ N_n : & trust\_me & & & N_n : & trust(C_n) \\ & \vdots & & & & \end{array}$$

for $n \geq 2$, $code(C_i) \in CLAUSE$, so that every instruction from the first column is immediately followed either by a clause proper, or by a nested chain[6] of the same form, while a *try* or *retry* instruction is always immediately followed by a *retry* or *trust*. We thus allow for *switching*—see next section, and [Warren 83, Ait-Kaci 91]— but do not assume any specific indexing method. A preferred method may be thought of as being abstractly encoded by the function *procdef*.

More precisely, our assumption may be expressed by using an auxiliary function $chain : CODEAREA \rightarrow CODEAREA^*$ such that

```
$chain(Ptr)$  =  if ( $code(Ptr) = try\_me\_else(N)$
                       or $code(Ptr) = retry\_me\_else(N)$ )
                 then $flatten([\,chain(Ptr+), chain(N)\,])$
                 elsif $code(Ptr) = trust\_me$
                 then $chain(Ptr+)$
                 elsif ( $code(Ptr) = try(C)$ or $code(Ptr) = retry(C)$ )
                 then $flatten([\,chain(C), chain(Ptr+)\,])$
                 elsif $code(Ptr) = trust(C)$
                 then $chain(C)$
                 else $[\,Ptr\,]$
```

Then our assumption may be exactly stated as

Compiler Assumption 1. The list $chain(procdef(G, Db))$ contains pointers to (code for) all candidate clauses for G in Db, in the right ordering.

The notion of 'code for' a clause means, for the time being, just the clause itself. The notions of *chain* and of 'code for' a clause will get refined in the sequel, leaving though the formulation of Compiler Assumption 1 valid, under a new interpretation.

Now the rules. *Goal* and *query success, cut, true, fail* are same as before, if $mode = Call$ is replaced in guards by $p = start$. The *Call* and *Enter* rules will also be of same form, if we additionally rewrite $mode := Call$ as $p := start$, and $mode = Enter$ as $code(p) \in CLAUSE$[7].

The interesting changes are to *Try* and *Retry* rules. It is now the task of the compiler to emit code which will avoid creating silly choicepoints, making the look-ahead guards of the previous section superfluous.

```
if $code(p) = try\_me\_else(N) \mid try(C)$
then push $temp$ with
    $store\_state\_in(temp)$
    $p(temp) := N \mid p(temp) := p+$
endpush
$p := p+ \mid p := C$
```

where | is the obvious notation for a pair of similar rules (with at most one alternative in the condition). It is now the task of the compiler to emit code which will not make existing choicepoints silly, replacing a *retry* instruction by the corresponding *trust* version.

```
if $code(p) = retry\_me\_else(N) \mid retry(C)$
then $fetch\_state\_from(b)$
     $restore\_cutpoint$
     $p(b) := N \mid p(b) := p+$
     $p := p+ \mid p := C$
```

```
if $code(p) = trust\_me \mid trust(C)$
then $fetch\_state\_from(b)$
     $restore\_cutpoint$
     $b := b(b)$
     $p := p+ \mid p := C$
```

[6] If the reader doesn't know the WAM, the next section might provide some idea of how nested chains can arise.

[7] The rules are spelled out in full in Appendix 2.

The proof of equivalence to the model of previous section would consist in straightforward verification, given Compiler Assumption 1, had we further assumed that the chains of indexing instructions are not nested, i.e. that *procdef* always points either to a flat chain of *try_me_else*, *retry_me_else*, *trust_me* instructions, each of them followed immediately by a single proper clause, or to a simple chain of *try*, *retry*, *trust* instructions.

We have however allowed for nested chains of indexing instructions as well. Note the new update *restore_cutpoint*. Of it we assume to restore somehow (magically) the value *ct* had when the current choicepoint was created, say by fetching the value having been stored on *try*. In case of flat chains, with at most one choicepoint per clause, our old update $ct := b(b)$ would do; with nested chains however there might be multiple choicepoints for a single clause, and $b(b)$ might be just one of them. Not wanting to commit ourselves to any specific way of 'implementing the *cut*', we shall rather rely on an abstract update, under

WAM Assumption 1. The effect of *restore_cutpoint* update is restoring *ct* to the value it had when the current choicepoint was created.

In case of nested chains the stack-to-stack component of the proof map should recognize multiple choicepoints and compress them to a single one (to be popped only when the outermost level gets popped). It would be straightforward to set up, for sake of proof, an intermediate model with additional *colored* indexing instructions used only for handling internal choicepoints, coloring the latter as well as internal, not unlike *internal_dynamic_else* used, for other purposes, by [Lindholm,O'Keefe 87]. The map from the intermediate model to the current one should simply forget the colors—the map to the model of the previous section could collapse colored choicepoints onto their nearest uncolored ancestors (interpreting colored instructions always as a $Retry_2$, except at the very beginning of a call, when a *try* followed by a sequence of colored *try*s should be interpreted as a *Try*, and at the very end, when a sequence of colored *trust*s followed by a *trust* should collapse to a $Retry_1$). These remarks should suffice to establish

Proposition 2.2. The model of 2.2. is, given Compiler Assumption 1 and WAM Assumption 1, correct and complete wrt that of 2.1.

2.3 Switching

We have so far allowed for indexing [Warren 83, Ait-Kaci 91] encapsulating it in the *procdef* function, which in a magical, i.e. abstract way finds a proper chain (see Compiler Assumption 1). Here we introduce the *switching instructions* [Warren 83, Ait-Kaci 91], permitting the compiler to arrange for a concrete indexing scheme, without however comitting ourselves (i.e. our correctness proof for the WAM) to any specific one.

The universe of instructions will be augmented by all instructions of form

$$switch\_on\_term(i, Lv, Lc, Ll, Ls), \quad switch\_on\_constant(i, N, T), \quad switch\_on\_structure(i, N, T)$$

with $i, N \in \mathcal{N}$ and $Lv, Lc, Ll, Ls, T \in CODEAREA$. The new instructions are 'executed' by rules

```
if $code(p) =$
    $switch\_on\_term(i, Lv, Lc, Ll, Ls)$
thenif $is\_var(x_i)$
then $p := Lv$
elsif $is\_const(x_i)$
then $p := Lc$
elsif $is\_list(x_i)$
then $p := Ll$
elsif $is\_struct(x_i)$
then $p := Ls$
```

```
if $code(p) = switch\_on\_constant(i, N, T)$
then $p := hash_c(T, N, x_i)$
```

```
if $code(p) = switch\_on\_structure(i, N, T)$
then $p := hash_s(T, N, funct(x_i), arity(x_i))$
```

where *funct, arity, arg* are the obvious term-analyzing functions, while *is_var, is_const, is_list, is_struct* the obvious recognizers of type, and $x_i \equiv arg(act, i)$. Hash table access is modelled by functions

$$\begin{aligned} hash_c &: CODEAREA \times \mathcal{N} \times ATOM \rightarrow CODEAREA \\ hash_s &: CODEAREA \times \mathcal{N} \times ATOM \times ARITY \rightarrow CODEAREA \end{aligned}$$

to be thought of as returning, for $hash_c(T, N, c)$ or $hash_s(T, N, f, a)$, the codepointer associated to c or f, a, respectively, in the hash table of size N located at T.

Our Compiler Assumption 1 now pertains to a refined *chain* function

```
$chain(Ptr)$ = if $code(Ptr) = switch\_on\_term(i, Lv, Lc, Ll, Ls)$
               then ( if $is\_var(x_i)$
                      then $chain(Lv)$
                      elsif $is\_const(x_i)$
                      then $chain(Lc)$
                      elsif $is\_list(x_i)$
                      then $chain(Ll)$
                      elsif $is\_struct(x_i)$
                      then $chain(Ls)$ )
               elsif $code(Ptr) = switch\_on\_constant(i, N, T)$
               then $chain(hash_c(T, N, x_i))$
               elsif $code(Ptr) = switch\_on\_structure(i, N, T)$
               then $chain(hash_s(T, N, funct(x_i), arity(x_i)))$
               else $old\_chain(Ptr)$
```

where *old_chain* is the *chain* function from 2.2.

Compiler Assumption 2. If switching is used, the *chain* function, occurring in Compiler Assumption 1, is to be interpreted as given above.

Given Compiler Assumption 2, inspection of rules (compared to definition of *chain*) suffices to establish

Switching Lemma. Switching preserves correctness and completeness.

Example 2. Given the Prolog program

$$\begin{array}{l} p(f_1(X)) :- q_1(X). \\ p(f_2(X)) :- q_2(X). \\ p(Other) :- default(Other). \end{array}$$

our assumption would allow at least the following two layouts of code (corresponding to the so called 'two–level' and 'one–level' indexing respectively), providing the same chain for any activator.

$$\begin{array}{ll}
 & try\_me\_else(C_3) \\
 & switch\_on\_term(1, C_1, fail, fail, L) \\
L: & switch\_on\_structure(1, 2, T) \\
 & \\
C_1: & try\_me\_else(C_2) \\
 & p(f_1(X)) :- q_1(X). \\
 & \\
C_2: & trust\_me \\
 & p(f_2(X)) :- q_2(X). \\
 & \\
C_3: & trust\_me \\
 & p(Other) :- default(Other). \\
 & \\
 & \text{with} \\
 & hash_s(T, 2, f_i, 1) = C_i+
\end{array}$$

$$\begin{array}{ll}
 & switch\_on\_term(1, C_1, C_3+, C_3+, L) \\
L: & switch\_on\_structure(1, 2, T) \\
 & \\
L_1: & try(C_1+) \\
 & trust(C_3+) \\
 & \\
L_2: & try(C_2+) \\
 & trust(C_3+) \\
 & \\
C_1: & try\_me\_else(C_2) \\
 & p(f_1(X)) :- q_1(X). \\
 & \\
C_2: & retry\_me\_else(C_3) \\
 & p(f_2(X)) :- q_2(X). \\
 & \\
C_3: & trust\_me \\
 & p(Other) :- default(Other). \\
 & \\
 & \text{with} \\
 & hash_s(T, 2, f_i, 1) = L_i
\end{array}$$

3 Clause Structure

In this section we analyse the compilation of clause structure into the WAM. Since the structure of clauses is largely independent of how terms and substitutions are represented, we keep the latter as abstract as possible, thus dealing with a simplified clause compilation scheme using only instructions for environment (de-)allocation, unification and calling.

3.1 Sharing the Continuation Stack

If the sequence of decorated goals, stored in *decglseq* and in choicepoints, is viewed as a stack, our model may be seen as a 'stack of stacks'. These stacks will necessarily contain plenty of common, copied structure.

Example 3. Take a Prolog program fragment

$$\begin{array}{l}
p :- r, s, t. \\
p :- u, v. \\
r :- w, x. \\
r.
\end{array}$$

with the initial query $[\,p, q\,]$. As the reader can easily verify, after three calls we shall have

$$\begin{array}{rcl}
p & = & procdef(w, db) \\
decglseq & = & [\,\langle\,[\,w, x\,], .\,\rangle, \langle\,[\,s, t\,], .\,\rangle, \langle\,[\,q\,], .\,\rangle\,] \\
decglseq(b) & = & [\,\langle\,[\,r, s, t\,], .\,\rangle, \langle\,[\,q\,], .\,\rangle\,]
\end{array}$$

where the (here irrelevant) cutpoints are all shown as dots.

Such wasteful copying of continuations can be (partially) avoided by sharing common pieces in a new data structure, the *environments*. In general, whenever a clause is entered, focusing, so to say, the attention on its body, a new environment is allocated, holding all the data necessary to continue the computation once all goals of that body are resolved. Parts (endsegments) of the environment chain will in general be shared between different choicepoints.

We then have no need to store the entire *decglseq* in a register (and in choicepoints)—the current *goal* will suffice. Once the activator is resolved with a clause, the rest of *goal* will be stored in a new environment as 'continuation goal' *cg*, and current *goal* will get the clause body. If the enviroments are properly stacked, once the body is exhausted, we just unload the stored *goal* and pop the environment. Since environments as well as cutpoints come with clause bodies, they are the logical place to store current *cutpt*.

What we need then is a universe *ENV* with functions

$$\begin{aligned} cg &: ENV \rightarrow GOAL \\ cutpt &: ENV \rightarrow STATE \\ ce &: ENV \rightarrow ENV \end{aligned}$$

where *cg* is the *continuation goal*, and *ce* links the enviroment stack (for *continuation environment*).

The role of (0−ary and unary) *decglseq* will now be taken over by

$$\begin{aligned} goal &\in GOAL & \qquad goal &: STATE \rightarrow GOAL \\ e &\in ENV & e &: STATE \rightarrow ENV \end{aligned}$$

with *goal* being the goal component of the first decorated goal of *decglseq*, while its cutpoint and continuation are contained in *e*.

Example 4. The sharing of continuation structure can already be seen if we redo the previous example. As the reader will soon be able to verify, we have, at the corresponding stage of computation,

$$\begin{aligned} p &= procdef(w, db) \\ goal &= [\, w, x \,] \\ cg(e) &= [\, s, t \,] \\ goal(b) &= [\, r, s, t \,] \\ cg(ce(e)) = cg(e(b)) &= [\, q \,] \end{aligned}$$

We would then have two stacks—of choicepoints and environments. For reasons to be explained below, they are usually represented in the WAM as *interleaved* on a single stack. To model this interleaving, we need a new superuniverse of both states and environments, with a stack-linking function and a common bottom

$$\begin{aligned} STACK &\supseteq STATE, ENV \\ - &: STACK \rightarrow STACK \\ bottom &\in STATE \cap ENV \end{aligned}$$

The old stack of choicepoints will always be reconstructible by following the *b* chain—as the stack of environments can be extracted by following the *ce* chain from *e*. The top of *STACK*, denoted as $tos(b,e)$, is their maximum (in sense of the ordering induced on $STACK$ by the linking function $-$).

In view of continuation sharing, we cannot instantiate the entire continuation with every unifying substitution. We shall instead keep all the goals properly renamed but uninstantiated—the current substitution will be applied to a literal only when it is to be resolved. We have then some new abbreviations:

$$\begin{aligned}
act &\equiv subres(fst(goal), s)\\
cont(l) &\equiv \textbf{if } l = bottom \textbf{ then } [\,]\\
&\ \ \textbf{else } append(cg(l), cont(ce(l)))\\
all\_done &\equiv goal = [\,]\ \&\ cont(e) = [\,]\\
succeed &\equiv goal := rest(goal)\\
cutpt &\equiv cutpt(e)
\end{aligned}$$

```
push $t$ with            $\equiv$   extend $STATE$ by $t$ with
  $updates(t)$                        $b := t$
endpush                               $b(t) := b$
                                      $t- := tos(b,e)$
                                      $updates(t)$
                                    endextend

allocate $t$ with        $\equiv$   extend $ENV$ by $t$ with
  $updates(t)$                        $e := t$
endalloc                              $ce(t) := e$
                                      $t- := tos(b,e)$
                                      $updates(t)$
                                    endextend
```

while the updates *store_state_in* and *fetch_state_from* now affect *goal, s, e* (minding that the old *decglseq* is now represented by *goal* and *e*). Under such a reading of abbreviations, all the rules except for *goal success* and *Enter* may be retained. The latter will take the form

```
if  $goal = [\,]$
    $\&\ NOT(all\_done)$
then  $goal := cg(e)$
      $e := ce(e)$
```

```
if $code(p) \in CLAUSE$
thenif $unify = nil$ then $backtrack$
else allocate $temp$ with
          $cg(temp) := rest(goal)$
          $cutpt(temp) := ct$
       endalloc
       $goal := bdy(clause)$
       $s := s \circ unify$
       $p := start$
       $vi := vi + 1$
where  $clause = rename(code(p), vi)$
         $unify = unify(act, hd(clause))$
```

Initial state will have $goal = query$ and $e = \langle [\,], bottom, bottom \rangle$. The reader may now verify the above example.

The proof map to the model of section 2.2 will be defined using two auxiliary functions (if all universes and functions of section 2 are marked with index 2)

$$F \ :\ STATE \rightarrow STATE_2$$

$$G \quad : \quad ENV \times SUBST \;\rightarrow\; (GOAL \times STATE_2)^*$$

such that

$$\begin{aligned}
F(bottom) &= bottom \\
G(bottom, S) &= [\,] \\
\\
decglseq(F(St)) &= [\,\langle\, subres(goal(St), s(St)), F(cutpt(e(St)))\,\rangle \mid G(e(St), s(St))\,] \\
p(F(St)) &= p(St) \\
b(F(St)) &= b(St) \\
& \ \text{for } St \neq bottom \\
\\
G(E, S) &= [\,\langle\, subres(cg(E), S), F(cutpt(ce(E)))\,\rangle \mid G(ce(E), S)\,] \\
& \ \text{for } E \neq bottom
\end{aligned}$$

The proof map $\mathcal{F}$ is then defined as leaving *db, stop, error, vi, s* as they are, reconstructing ct_2 as $F(ct)$, while $decglseq_2$, p_2 and b_2 are defined as components of $F(\langle\, g, p, s, e, b\,\rangle)$. On rules $\mathcal{F}$ is the obvious homonymy, and its commutativity with execution of the rules is now straightforward to verify, yielding

Proposition 3.1. The model of 3.1. is correct and complete wrt that of 2.2.

Had we not interleaved the two stacks, of choicepoints and environments, we would sooner or later have to face the following problem. Due to the possibility of multiple choicepoints for the same call, and to some optimizations to be introduced later, it is quite possible for more than one choicepoint to rely on (i.e. to point with e to) the same environment. With a more realistic representation of stacks, where a 'popped' item may get irretrievably lost (discarded or overwritten), how are we to be sure that the environment is still there whenever a choicepoint needs it?

If the stacks are interleaved as above, the problem cannot occur as long as the stack discipline is maintained, since every choicepoint *hides* its environment by being above it on the stack. Straightforward induction namely establishes

Hiding Lemma. For any choicepoint $l \in STATE$, $e(l) < l$, in terms of the ordering induced on $STACK$ by $-$.

3.2 Compilation of Clause Structure

Although much of continuation structure is by now shared in the environment stack, the clause bodies (and their end-segments), in the role of *goal*, still get wastefully copied to and from both environments and choicepoints. We might, more efficiently, store and copy just 'positions in bodies', since the clauses have to be somehow represented in the program. In other words, we might *represent clauses by code*, i.e. *compile* them. We assume a function

$$\begin{aligned}
compile \quad &: \quad CLAUSE \;\rightarrow\; INSTR^* \\
compile(H :- G_1, \ldots, G_n) \quad &\equiv \quad [\; allocate, unify(H), \\
& \qquad\ \ call(G_1), \ldots, call(G_n), \\
& \qquad\ \ deallocate, proceed\,]
\end{aligned}$$

where the universe *INSTR* is extended to contain

$$\{\ allocate, deallocate, unify(G), call(G), proceed \mid G \in LIT\ \}$$

This is still a very abstract notion of compilation—Prolog goals and terms are still present inside instructions as they are, 'unrepresented'. Their representation will be dealt with in section 4.

Our notion of 'code for' a clause, from Compiler Assumption 1, has to evolve now. It may be clearly stated using an auxiliary function

$$\begin{array}{rcl} unload & : & CODEAREA \rightarrow INSTR^* \\ unload(Ptr) & \equiv & \textbf{if}\ code(Ptr) = proceed \\ & & \textbf{then}\ [\,proceed\,] \\ & & \textbf{else}\ [\,code(Ptr) \mid unload(Ptr+)\,] \end{array}$$

suggesting that sequences of instructions had been loaded into codearea. We shall then say that $Ptr \in CODEAREA$ *points to code for a clause* Cl if

$$unload(Ptr) = compile(Cl).$$

Compiler Assumption 3. The notion of 'code for' a clause, in Compiler Assumption 1, is to be interpreted as above.

The wording of the above definition will survive, although the definition of *compile* will, in section 4, evolve.

The structure of clause bodies is represented in *CODEAREA* by $+$ function. We shall also need its inverse $-$. What we shall not need any more is explicit presence of *goal*s in registers, environments and choicepoints. The *cg* function will now be replaced by a *continuation pointer* cp, with two homonyms representing current and backtracking *goal*,

$$\begin{array}{rcl} cp & : & ENV \rightarrow CODEAREA \\ cp & \in & CODEAREA \\ cp & : & STATE \rightarrow CODEAREA \end{array}$$

where the value of the last one should be *store*d and *fetch*ed by choicepoint handling updates. We shall maintain the

Continuation Pointer Constraint. Whenever $NOT(all\_done)$, $code(cp-)$ is of form $call(G)$.

The literal currently being resolved will then at all times be accessible via $cp-$. It means that the current *goal* will be reconstructible as the sequence of all literals *call*ed in $unload(cp-)$, properly renamed, i.e. by using a function

$$\begin{array}{rcl} g & : & CODEAREA \times \mathcal{N} \rightarrow GOAL \\ g(Ptr, i) & \equiv & \textbf{if}\ code(Ptr) = proceed\ \textbf{then}\ [\,] \\ & & \textbf{elsif}\ code(Ptr) = call(G)\ \textbf{then}\ [\,rename(G, i) \mid g(Ptr+, i)\,] \\ & & \textbf{else}\ g(Ptr+, i) \end{array}$$

Proper renaming requires some care now. In the last section bodies have been immediately renamed as soon as they entered the computation. Since (codes for) bodies are now shared between different calls, we have to keep the proper renaming index wherever the identity of a call is maintained, and that is the environment. We need then a function

$$vi \quad : \quad ENV \rightarrow \mathcal{N}$$

Elements of the old signature will then be reconstructible as

$$\begin{aligned} goal &\equiv g(cp-, vi(e)) \\ act &\equiv subres(fst(goal), s) \\ goal(b) &\equiv g(cp(b)-, vi(e(b))) \\ cg(e) &\equiv g(cp(e), vi(ce(e))) \end{aligned}$$

Under this representation, two abbreviations must change.

$$\begin{aligned} all\_done &\equiv code(p) = proceed \ \& \ code(cp) = proceed \\ succeed &\equiv p := p+ \end{aligned}$$

In the **initial state** with query $[\,G_1, \ldots, G_n\,]$ we must assume, in view of Continuation Pointer Constraint,

$$\begin{aligned} unload(p) &= [\,call(G_1), \ldots, call(G_n), proceed\,] \\ cp &= p+ \end{aligned}$$

with the initial environment storing the initial value of *vi* and *bottom* cutpoint.

Transcribing further $p = start$ to $code(p) = call(G)$ we can retain all rules except *Call, goal success, Enter*. *Call* rule[8] must maintain the Continuation Pointer Constraint, and, in addition to setting *p* and *ct* as before, set $cp := p+$. The action of *goal success, Enter* gets decomposed by new instructions into more primitive steps, and these rules are replaced by

```
if $code(p) = deallocate$
then $e := ce(e)$
     $cp := cp(e)$
     $succeed$

if $code(p) = allocate$
then allocate $temp$ with
        $cp(temp) := cp$
        $vi(temp) := vi$
        $cutpt(temp) := ct$
     endalloc
     $succeed$
```

```
if  $code(p) = proceed$
    $\& \ NOT(all\_done)$
then $p := cp$

if $code(p) = unify(H)$
thenif $unify = nil$
then $backtrack$
else  $s := s \circ unify$
      $vi := vi + 1$
      $succeed$
where
   $unify = unify(act, rename(H, vi))$
```

Note that global *vi* is used for renaming the clause head, while the definition of *act* above uses $vi(e)$.

Example 5. Reproducing Example 3 in our current model, we would (initially) have the following

[8] The rules are spelled out in full in Appendix 3.

layout of code.

```
p :   call(p)        C1 :   allocate        C2 :   allocate
      call(q)               unify(p)               unify(p)
      proceed               call(r)                call(u)
                            call(s)                call(v)
                            call(t)                deallocate
                            deallocate             proceed
                            proceed

                     C3 :   allocate        C4 :   allocate
                            unify(r)               unify(r)
                            call(w)                deallocate
                            call(x)                proceed
                            deallocate
                            proceed
```

As the reader can easily verify, after three calls we would have the following situation:

$$\begin{aligned} p &= \mathit{procdef}(w, db) \\ cp &= C_3 + 3 \\ cp(e) = cp(b) &= C_1 + 3 \\ cp(ce(e)) = cp(e(b)) &= p_{init} + 1 \end{aligned}$$

Note that, although e and b have the same value of cp, the values of $cg(e)$ and $goal(b)$ are different, as they should be.

The state component of the proof map $\mathcal{F}$ has essentially been given above. The rule map is homonymous except for

$$\begin{aligned} \mathcal{F}([\,\mathit{deallocate}, \mathit{proceed}\,]) &= [\,\mathit{goal\ success}\,] \\ \mathcal{F}([\,\mathit{allocate}, \mathit{unify}(H)\,]) &= [\,\mathit{Enter}\,] \end{aligned}$$

Maintenance of Continuation Pointer Constraint is straightforward to verify, by inspection of rules. Commutativity of $\mathcal{F}$ with rule execution then proves

Proposition 3.2. The model of 3.2 is correct and complete wrt that of 3.1.

4 Term Structure

In this section we refine our model to allow for representation of terms and substitutions. We introduce stepwise the heap, the push-down-list and unification, put- and get-instructions and trailing, assuming in these steps (for sake of simplicity and analysis) that all variables are permanent and get initialized to *unbound* as soon as they are allocated. This allows us to introduce, in a further refinement step, the fine points of the WAM as local optimizations and related correctness preseving devices—they are known as environment trimming, local and unsafe values, last call optimization, Warren's variable classification and on-the-fly-initialization. We conclude with a treatment of the cut.

4.1 Representing Terms—Compilation of Term Structure

WAM representation of terms will be modelled by a structure of 'locations'

$$(DATAAREA; +, -; val),$$

where

$$+, - \quad : \quad DATAAREA \rightarrow DATAAREA$$

are mutually inverse functions which may be viewed as 'organizing the locations', while their 'contents' are accessible via

$$val \quad : \quad DATAAREA \rightarrow PO + MEMORY$$

where PO (for *Prolog Objects*) is a universe supplied with functions

$$\begin{aligned} type \quad &: \quad PO \rightarrow \{Ref, Const, List, Struct, Funct\} \\ ref \quad &: \quad PO \rightarrow ATOM + DATAAREA + ATOM \times ARITY \end{aligned}$$

where $ARITY = \{0, \ldots, maxarity\}$ and $MEMORY$ is a universe containing $DATAAREA$, to be elaborated and used below. Its role in the codomain of val is to enable storage of pure pointers in $DATAAREA$.

We shall abbreviate $type(val(l)), ref(val(l))$ as $type(l), ref(l)$ respectively, for $l \in DATAAREA$. Assignment of values, (very much) distinct ¿from assignment of pointers, will be abbreviated as

$$l_1 \leftarrow l_2 \;\equiv\; val(l_1) := val(l_2).$$

Further abbreviations that will be used often are

$$\begin{aligned} l \leftarrow \langle T, R \rangle \quad &\equiv \quad type(l) := T \\ & \quad ref(l) := R \\ unbound(l) \quad &\equiv \quad (\, type(l) = Ref \;\&\; ref(l) = l) \\ mk\_ref(l) \quad &\equiv \quad \langle\, Ref, l \,\rangle \\ mk\_unbound(l) \quad &\equiv \quad l \leftarrow mk\_ref(l). \end{aligned}$$

We shall freely use the (partial) arithmetics and ordering induced on $DATAAREA$ by the successor function $+$; they are partial since we do not assume of $DATAAREA$ to be Archimedean—cf. discussion in Section 4.2. We shall further assume (partial) functions

$$\begin{aligned} deref \quad &: \quad DATAAREA \rightarrow DATAAREA \\ term \quad &: \quad DATAAREA \rightarrow TERM \end{aligned}$$

such that $deref(l)$ follows the reference chain from l (composing possible variable–to–variable bindings coming from different substitutions), while $term(l)$ reconstructs the term represented at 'location' l. More precisely, we assume

$$deref(l) \quad = \quad \begin{cases} deref(ref(l)) & \text{if } type(l) = Ref \;\&\; NOT(unbound(l)) \\ l & \text{otherwise} \end{cases}$$

$$term(l) \quad = \quad \begin{cases} mk\_var(l) & \text{if } unbound(l) \\ term(deref(l)) & \text{if } type(l) = Ref \;\&\; NOT(unbound(l)) \\ ref(l) & \text{if } type(l) = Const \\ [\, term(ref(l)) \mid term(ref(l){+}) \,] & \text{if } type(l) = List \\ f(a_1, \ldots, a_n) & \text{if } \begin{cases} type(l) = Struct \\ ref(ref(l)) = \langle f, n \rangle \\ term(ref(l) + i) = a_i \end{cases} \end{cases}$$

Of *mk_var* we assume to associate a unique Prolog variable to an arbitrary location in *DATAAREA*.

Example 6. The term $s(Y_2, [\,c \mid Y_1\,]))$ would for instance be represented by a location l such that

$$\begin{array}{lll} l = \langle\, \mathit{Struct}, l_1 \,\rangle & l_1 = \langle\, \mathit{Funct}, \langle s, 2\rangle \,\rangle & l_1{+} = \langle\, \mathit{Ref}, l_1{+} \,\rangle \\ l_1 {+}{+} = \langle\, \mathit{List}, l_2 \,\rangle & l_2 = \langle\, \mathit{Const}, c, \rangle & l_2{+} = \langle\, \mathit{Ref}, l_2{+} \,\rangle \end{array}$$

The reader may draw a picture and verify $\mathit{term}(l)$. It might be instructive to forget for a moment about special representation of constants and lists, and redo the example, viewing constants and lists as $0-$ary and binary structures, respectively.

Remark. The condition $\mathit{term}(l) \in \mathit{TERM}$, to be used often below, implies the following properties.

$$\begin{array}{llll} \mathit{if} & \mathit{type}(l) = \mathit{Const} & \mathit{then} & \mathit{ref}(l) \in \mathit{ATOM} \\ \mathit{if} & \mathit{type}(l) \in \{\mathit{Ref}, \mathit{List}, \mathit{Struct}\} & \mathit{then} & \mathit{ref}(l) \in \mathit{DATAAREA} \\ & & & \mathit{term}(l) \in \mathit{TERM} \\ \mathit{if} & \mathit{type}(l) = \mathit{List} & \mathit{then} & \mathit{term}(\mathit{ref}(l){+}) \in \mathit{TERM} \\ \mathit{if} & \mathit{type}(l) = \mathit{Struct} & \mathit{then} & \mathit{type}(\mathit{ref}(l)) = \mathit{Funct} \\ & & & \mathit{ref}(\mathit{ref}(l)) = \langle f, n\rangle \\ & & & \mathit{term}(\mathit{ref}(l) + i) \in \mathit{TERM} \end{array}$$

where, in the last condition, $\langle f, n\rangle \in \mathit{ATOM} \times \mathit{ARITY}$, $i = 1, \ldots, n$.

4.1.1 The Heap, the Push-Down List and Unification

Prolog structures will be represented on the *HEAP*, a subalgebra of *DATAAREA*

$$(\mathit{HEAP}; h, \mathit{boh}; +, -; \mathit{val})$$

to be used as a stack, with $h, \mathit{boh} \in \mathit{HEAP}$ representing top and bottom, $\mathit{str} \in \mathit{HEAP}$ the subterm (or structure) pointer to be used for navigating through substructures. The active part of the *HEAP* will be abbreviated as $\mathit{heap} \equiv \{l \in \mathit{HEAP} \mid \mathit{boh} \leq l < h\}$ (its finiteness follows from *boh* being the initial value of h), and locations $l \in \mathit{heap}$ such that $\mathit{type}(l) = \mathit{Ref}$ will be referred to as *heap variables*.

Goal arguments will be represented in another subdomain *AREGS* of *DATAAREA*, disjoint from the *HEAP*, on which we shall never use $+, -$; we assume just a function $x : \mathcal{N} \rightarrow \mathit{AREGS}$, $x_i \equiv x(i)$.

For unification we shall use a dedicated stack, the *pushdown list*, represented by an algebra

$$(\mathit{PDL}, \mathit{DATAAREA}; \mathit{pdl}, \mathit{nil}; +, -; \mathit{ref}')$$

which may but need not be seen as embedded in *DATAAREA*, with $\mathit{left} \equiv \mathit{ref}'(\mathit{pdl})$, $\mathit{right} \equiv \mathit{ref}'(\mathit{pdl}-)$.

In the unification algorithm below we shall use an abstract update $\mathit{bind}(l_1, l_2)$, of which we assume

WAM Assumption 2. For any $l, l_1, l_2 \in \mathit{DATAAREA}$ such that

- $\mathit{unbound}(l_1)$
- the variable $\mathit{mk\_var}(l_1)$ does not occur in $\mathit{term}(l_2)$,

if $term, term'$ are the values of $term(l)$ before and after the execution of $bind(l_1, l_2)$ respectively, and s is the substitution associating $term(l_2)$ to $mk\_var(l_1)$, then $term' = subres(term, s)$.

The mathematical (nondeterministic) unification algorithm [Apt 90, Lloyd 84] should fail on attempt to bind a variable to a term in which it occurs—unification is then said to be STO (subject to *occur-check*). Had we chosen to model such *unification with occur-check*, we would simply require of $bind(l_1, l_2)$ to trigger *backtracking* whenever the second condition of WAM Assumption 2 is violated, i.e. when $mk\_var(l_1)$ occurs in $term(l_2)$.

We choose however to allow for the usual practice of Prolog implementations, which, for pragmatic reasons, generally skip the occur check, and to adhere to the draft standard proposal [ISO WG17 93], which refrains from specifying the behaviour of systems upon STO unification by considering it as 'implementation dependent'. Hence we make no assumption whatsoever on STO binding. This decision is necessarily reflected in Unification Lemma and Pure Prolog Theorem below, which must refrain from stating anything for this case.

Unification will be triggered by setting a special 0−ary function *what_to_do* to *Unify*, given that the terms to be unified have already been pushed to *PDL*. We then have $what\_to\_do \in \{ Run, Unify \}$, and the following Unification Rule, using *dl, dr* as abbreviations for $deref(left)$, *deref*(*right*) respectively:

```
if $OK$ & $what\_to\_do = Unify$
thenif $pdl = nil$
then $what\_to\_do := Run$
elsif $unbound(dl)$
then $bind(dl, dr)$
        $pdl := pdl --$
elsif $unbound(dr)$
then $bind(dr, dl)$
        $pdl := pdl --$
elsif $type(dl) \neq type(dr)$
then $backtrack$
        $what\_to\_do := Run$
elsif $type(dl) = Const$
thenif $ref(dl) = ref(dr)$
then $pdl := pdl --$
else $backtrack$
       $what\_to\_do := Run$
elsif $type(dl) = List$
then $ref'(pdl-) := ref(dr)$
        $ref'(pdl) := ref(dl)$
        $ref'(pdl+) := ref(dr)+$
        $ref'(pdl++) := ref(dl)+$
        $pdl := pdl ++$
elsif $same\_funcs(dl, dr)$
then seq
          $pdl := pdl --$
          seq $i = 1, \ldots, arity(ref(dl))$
            $ref'(pdl+) := ref(dr) + i$
            $ref'(pdl++) := ref(dl) + i$
            $pdl := pdl ++$
         endseq
       endseq
else $backtrack$
       $what\_to\_do := Run$
```

where *same_funcs*(l_1, l_2) stands for $ref(ref(l_1)) = ref(ref(l_2))$, given that both are of type *Struct*.

Unification will be invoked exclusively by executing *unify* update, where

$$\begin{array}{lcl} unify(l_1, l_2) & \equiv & ref'(nil+) := l_2 \\ & & ref'(nil++) := l_1 \\ & & pdl := nil++ \\ & & what\_to\_do := Unify \end{array}$$

The above has the flavour of assembly code for calling a (recursive) subroutine, pushing the arguments to a stack, with setting *what_to_do* to *Unify* playing the role of 'jump to subroutine', and resetting it to *Run* that of 'return from subroutine'. Of other rules in the sequel, 'running the main program', we shall tacitly assume to contain the guard OK & $what\_to\_do = Run$.

In view of WAM Assumption 2, it is straightforward to recognize the above as a deterministic variant of Herbrand's algorithm [Apt 90, Ait-Kaci 91]—thus

Unification Lemma. If $term(l_1), term(l_2) \in TERM$, then the effect of executing $unify(l_1, l_2)$, for any $l \in DATAAREA$ such that $term(l) \in TERM$, is as follows:

- if s is the most general unifier of $term(l_1)$, $term(l_2)$, the new value of $term(l)$, when $what\_to\_do$ is set to Run again, will be the result of applying s to its old value;
- if the abstract unification algorithm fails without invoking the occur-check, *backtrack* update will be executed.

Had we assumed of *bind* to involve the occur-check, we could drop the non-STO limitations from our lemma. If the reader prefers a different unification algorithm, and hence finds our description to be overspecific, he is welcome to consider the Unification Lemma as an additional WAM Assumption, assuming further that the *unify* update binds variables by invoking the abstract *bind* update—all results (and proofs) below will remain valid.

4.1.2 Putting

Here we develop (almost) WAM code for constructing body goals. We shall use

$$(CODEAREA; p, bottom; +, -; code),$$

with $MEMORY \supseteq DATAAREA + CODEAREA$. The universe $INSTR$ of instructions is assumed to contain all instructions of form

$$\begin{array}{lll}
put\_value(y_n, x_j), & put\_constant(c, x_j), & put\_list(x_j), \\
put\_structure(f, a, x_i), & unify\_value(x_n), & unify\_value(y_n), \\
unify\_constant(c), & get\_value(y_n, x_j), & get\_constant(c, x_j), \\
get\_list(x_j), & get\_structure(f, a, x_j), & unify\_variable(x_n)
\end{array}$$

with $n, j, i \in \mathcal{N}$, $c, f \in ATOM$, $a \in ARITY$, $y_n \in DATAAREA$, and will be tacitly extended with any further instructions occurring below.

To define the sequence of putting instructions corresponding to a body goal, we rely on the notion of *term normal form* of first order logic, which comes in two variants, corresponding to analysis and synthesis of terms.

$$\begin{array}{rcl}
nf(X_i = Y_n) & = & [X_i = Y_n] \\
nf(X_i = c) & = & [X_i = c] \\
nf(Y_i = Y_n) = nf(c = c) & = & [\,] \\
nf_s(X_i = [s_1 \mid s_2]) & = & flatten([nf_s(Z_1 = s_1), nf_s(Z_2 = s_2), X_i = [Z_1 \mid Z_2]]) \\
nf_s(X_i = f(s_1, \ldots, s_m)) & = & flatten([nf_s(Z_1 = s_1), \ldots, nf_s(Z_m = s_m), \\
 & & \qquad X_i = f(Z_1, \ldots, Z_m)]) \\
nf_a(X_i = [s_1 \mid s_2]) & = & flatten([X_i = [Z_1 \mid Z_2], nf_a(Z_1 = s_1), nf_a(Z_2 = s_2)]) \\
nf_a(X_i = f(s_1, \ldots, s_m)) & = & flatten([X_i = f(Z_1, \ldots, Z_m), \\
 & & \qquad nf_a(Z_1 = s_1), \ldots, nf_a(Z_m = s_m)])
\end{array}$$

where Z_i is a fresh X–variable if s_i is a list or a structure, s_i otherwise.

By the term normal form theorem of first order logic the equation $X = t$ is equivalent to the set of equations $nf(X = t)$ representing its normal form, which is computationally reflected by the (Prolog) effect of executing all members of $nf(X = t)$ being the same as that of executing $X = t$, both in case of analysis and of synthesis.

Example 7. For instance, we might have

$$\begin{array}{rcl}
\mathit{nf}_s(X_2 = s(Y_2, [\,c \mid Y_1\,]) & = & \mathit{flatten}([[\,], \mathit{nf}_s(X_3 = [\,c \mid Y_1\,]), X_2 = s(Y_2, X_3)]) \\
& = & [\,X_3 = [\,c \mid Y_1\,], X_2 = s(Y_2, X_3)\,] \\
\mathit{nf}_a(X_2 = s(Y_2, [\,c \mid Y_1\,]) & = & \mathit{flatten}([\,X_2 = s(Y_2, X_3), [\,], \mathit{nf}_a(X_3 = [\,c \mid Y_1\,])]) \\
& = & [\,X_2 = s(Y_2, X_3), X_3 = [\,c \mid Y_1\,]\,]
\end{array}$$

The *put_instr* of a normalized equation will be according to the following table, where j stands for an arbitrary 'top level' index (corresponding to input $X_i = t$ for term normalization), k for a 'non top level' index (corresponding to an auxiliary variable introduced by normalization itself), and i for any index whatsoever:

$$\begin{array}{rcl}
X_j = Y_n & \longrightarrow & [\,\mathit{put\_value}(y_n, x_j)\,] \\
X_j = c & \longrightarrow & [\,\mathit{put\_constant}(c, x_j)\,] \\
X_i = [\,Z_1 \mid Z_2\,] & \longrightarrow & [\,\mathit{put\_list}(x_i), \mathit{unify}(z_1), \mathit{unify}(z_2)\,] \\
X_i = f(Z_1, \ldots, Z_a) & \longrightarrow & [\,\mathit{put\_structure}(f, a, x_i), \\
& & \quad \mathit{unify}(z_1), \ldots, \mathit{unify}(z_a)\,]
\end{array}$$

with $y_i \in \mathit{DATAAREA}$, $x_i \in \mathit{AREGS}$, and

$$\mathit{unify}(z_i) \quad = \quad \begin{cases} \mathit{unify\_value}(y_n) & \text{if } Z_i = Y_n \\ \mathit{unify\_value}(x_k) & \text{if } Z_i = X_k \\ \mathit{unify\_constant}(c) & \text{if } Z_i = c \end{cases}$$

Note that, by definition of nf_s, equations of form $X_k = Y_n$ or $X_k = c$ can never occur there. For correspondence between Prolog variables X, Y, Z and $x, y, z \in \mathit{DATAAREA}$ see Putting Lemma below.

The function *put_code* is defined by flattening the result of mapping *put_instr* along $\mathit{nf}_s(X_i = t)$. It is auxiliary to *put_seq*, specifying how a body goal is to be compiled:

$$\mathit{put\_seq}(g(s_1, \ldots, s_m)) = \mathit{flatten}(\,[\,\mathit{put\_code}(X_1 = s_1), \ldots, \mathit{put\_code}(X_m = s_m)\,]\,)$$

with 'top level' $j = 1, \ldots, m$. The reader knowledgeable about the WAM will notice that this is not quite the WAM code, but wait till section 4.3.

Example 8. A body goal of form $p(Y_1, s(Y_2, [\,c \mid Y_1\,]))$ would for instance generate the following code:

$$\begin{array}{rl}
\mathit{put\_seq}(p(Y_1, s(Y_2, [\,c \mid Y_1\,]))) = [& \mathit{put\_value}(y_1, x_1), \mathit{put\_list}(x_3), \mathit{unify\_constant}(c), \\
& \mathit{unify\_value}(y_1), \mathit{put\_structure}(s, 2, x_2), \mathit{unify\_value}(y_2), \\
& \mathit{unify\_value}(x_3)\,]
\end{array}$$

Putting code will be 'executed' by the following rules (ignoring, for now, the value of *mode*, but see the next subsection).

```
if $code(p) = put\_value(l, x_j)$
then $x_j \leftarrow l$
     $succeed$
```

```
if $code(p) = put\_constant(c, x_j)$
then $x_j \leftarrow \langle Const, c \rangle$
     $succeed$
```

```
if $code(p) = put\_list(x_i)$
then $h \leftarrow \langle List, h+ \rangle$
     $x_i \leftarrow \langle List, h+ \rangle$
     $h := h+$
     $mode := Write$
     $succeed$
```

```
if $code(p) = unify\_value(l)$
   & $mode = Write$
then $h \leftarrow l$
     $h := h+$
     $succeed$
```

```
if $code(p) = unify\_constant(c)$
   & $mode = Write$
then $h \leftarrow \langle Const, c \rangle$
     $h := h+$
     $succeed$
```

```
if $code(p) = put\_structure(f, a, x_i)$
then $h \leftarrow \langle Struct, h+ \rangle$
     $h+ \leftarrow \langle Funct, \langle f, a \rangle \rangle$
     $x_i \leftarrow \langle Struct, h+ \rangle$
     $h := h++$
     $mode := Write$
     $succeed$
```

Then we have

Putting Lemma. If all variables occurring in Prolog literal $g(t_1, \ldots, t_m)$ are among $\{Y_1, \ldots, Y_l\}$, and if, for $y_n \in DATAAREA$ with $term(y_n) \in TERM$, s the substitution associating every Y_n with $term(y_n)$, $n = 1, \ldots, l$, and X_i fresh pairwise distinct variables, $i = 1, \ldots, m$,

then the effect of executing (setting p to a value where *unload* yields) $put\_seq(g(t_1, \ldots, t_m))$ is that

$term(x_i)$ gets value of $subres(t_i, s)$, $i = 1, \ldots, m$.

Proof by induction over cumulative size of terms. Note that substructure representing X_k variables, generated by nf_s, always get 'instantiated', i.e. $term(x_k)$ gets a value in $TERM$, by a *put*, before being 'used' by a *unify*. Uninstantiated variables inside lists and structures get represented on the heap. In case of unbound Y_n, associating it to $mk\_var(y_n)$ amounts to renaming.

The reader might verify the effect of executing the code of Example 8.

4.1.3 Getting

The compilation of clause head, *compile time unification*, will again be specified by an abstract function *get_seq*, associating a sequence of instructions to a Prolog literal, relying on term normal forms. The *get_instr* of a normalized equation is defined by a table similar to that for *put_instr*, under the same conventions.

$$\begin{array}{rcl}
X_j = Y_n & \rightarrow & [\, get\_value(y_n, x_j)\,] \\
X_j = c & \rightarrow & [\, get\_constant(c, x_j)\,] \\
X_i = [\, Z_1 \mid Z_2\,] & \rightarrow & [\, get\_list(x_i), unify(z_1), unify(z_2)\,] \\
X_i = f(Z_1, \ldots, Z_a) & \rightarrow & [\; get\_structure(f, a, x_i), \\
 & & \quad unify(z_1), \ldots, unify(z_a)\,]
\end{array}$$

where now

$$unify(z_i) = \begin{cases} unify\_value(y_n) & if\ Z_i = Y_n \\ unify\_variable(x_k) & if\ Z_i = X_k \\ unify\_constant(c) & if\ Z_i = c \end{cases}$$

By definition of nf_a, equations of form $X_k = Y_n$ or $X_k = c$ can never occur there.

The function *get_code* is defined by flattening the result of mapping *get_instr* along $nf_a(X_i = t)$. It is auxiliary to *get_seq*, specifying how a clause head is to be compiled:

$$get\_seq(g(s_1, \ldots, s_m)) = flatten(\,[\,get\_code(X_1 = s_1), \ldots, get\_code(X_m = s_m)\,]\,)$$

with 'top level' $j = 1, \ldots, m$.

Example 9. A clause head of form $p(Y_1, s(Y_2, [\,c \mid Y_1\,]))$ might generate the following code.

$$\begin{aligned} get\_seq(p(Y_1, s(Y_2, [\,c \mid Y_1\,]))) = [\ & get\_value(y_1, x_1), get\_structure(s, 2, x_2), unify\_value(y_2), \\ & unify\_variable(x_3), get\_list(x_3), unify\_constant(c), \\ & unify\_value(y_1)\,] \end{aligned}$$

The *unify* instructions, corresponding to arguments of structures and lists, will now have two roles to play, depending on whether a structure or a list in clause head has to be constructed on the heap (when unified with an unbound variable) or just needs to be matched with an existing structure. The roles will be distinguished by a 0–ary function *mode*, taking values in $\{$ *Read, Write*$\}$. Note that, when used for putting, the *unify* instructions needed to fulfill only the first role, and hence only the *Write* value of *mode* was present.

The getting code sequence will be then executed by the following (additional) rules.

```
if $code(p) = get\_value(l, x_j)$
then $unify(l, x_j)$
     $succeed$
```

```
if $code(p) = unify\_value(l)$
   & $mode = Read$
then $unify(l, str)$
     $str := str+$
     $succeed$
```

```
if $code(p) = unify\_variable(l)$
   & $mode = Read$
then $l \leftarrow str$
     $str := str+$
     $succeed$
```

```
if $code(p) = unify\_variable(l)$
   & $mode = Write$
then $mk\_heap\_var(l)$
     $succeed$
```

```
if $code(p) = get\_constant(c, x_i)$
thenif $type(deref(x_i)) = Ref$
then $deref(x_i) \leftarrow \langle Const, c \rangle$
     $trail(deref(x_i))$
     $succeed$
elsif $deref(x_i) = \langle Const, c \rangle$
then $succeed$
else $backtrack$
```

```
if $code(p) = unify\_constant(c)$
   & $mode = Read$
thenif $type(deref(str)) = Ref$
then $deref(str) \leftarrow \langle Const, c \rangle$
     $trail(deref(str))$
     $str := str+$
     $succeed$
elsif $deref(str) = \langle Const, c \rangle$
then $str := str+$
     $succeed$
else $backtrack$
```

```
if $code(p) = get\_list(x_i)$
thenif $type(deref(x_i)) = Ref$
then $deref(x_i) \leftarrow \langle List, h+ \rangle$
     $trail(deref(x_i))$
     $h \leftarrow \langle List, h+ \rangle$
     $h := h+$
     $mode := Write$
     $succeed$
elsif $type(deref(x_i)) = List$
then $str \leftarrow ref(deref(x_i))$
     $mode := Read$
     $succeed$
else $backtrack$
```

```
if $code(p) = get\_structure(f, a, x_i)$
thenif $type(deref(x_i)) = Ref$
then $deref(x_i) \leftarrow \langle Struct, h+ \rangle$
     $trail(deref(x_i))$
     $h \leftarrow \langle Struct, h+ \rangle$
     $h+ \leftarrow \langle Funct, \langle f, a, \rangle \rangle$
     $h := h++$
     $mode := Write$
     $succeed$
elsif $type(deref(x_i)) = Struct$
     $\& \; ref(ref(deref(x_i))) = \langle f, a \rangle$
then $str := ref(deref(x_i))+$
     $mode := Read$
     $succeed$
else $backtrack$
```

where

```
$mk\_heap\_var(l) \equiv mk\_unbound(h)$ .
                        $l \leftarrow mk\_ref(h)$
                        $h := h+$
```

Note that *unify_variable* in *Write* mode has, at present, to allocate a fresh heap variable to be subsequently bound by a *get_list* or *get_structure* instruction (in 4.3.1. it will assume another role as well). Then we have

Getting Lemma.If all variables occurring in Prolog literal $g(t_1, \ldots, t_m)$ are among $\{Y_1, \ldots, Y_l\}$, and if further $y_n \in DATAAREA$ with $unbound(y_n)$, $n = 1, \ldots, l$, X_i fresh pairwise distinct variables with $x_i \in DATAAREA$ and $term(x_i) \in TERM$, $1 \leq i \leq m$, and

a) s is the unifying substitution of t_i and $term(x_i)$, $1 \leq i \leq m$, or

b) such unification fails without invoking the occur-check,

then the effect of executing (setting program pointer p to) $load(get\_seq(g(t_1, \ldots, t_m)))$ for any $l \in DATAAREA$ with $term(l) = t \in TERM$ (before execution), is respectively

a) $term(l)$ gets the value of $subres(t, s)$ (up to renaming of Y's), or

b) backtracking.

Proof by induction over cumulative size of terms, relying on Unification Lemma. Note that substructure descriptors X_k, generated by nf_a, always get represented on the heap (and $term(x_k)$ gets a value in $TERM$) by *unify_variable*, before being 'used' by *get_structure* or *get_list*.

The reader might verify the effect of executing the code of Example 9.

Since the heap will, in the sequel, turn out to be the most persistent of all data areas (subdomains of $DATAAREA$), we shall wish to uphold the following constraint:

Heap Variables Constraint. No heap variable points outside the heap, i.e. for any $l \in heap$, if $type(l) = Ref$, then $ref(l) \in heap$.

If we assume that the abstract *bind* update does not violate the constraint (in case of two unbound variables, since structures are constructed on the heap anyway), a simple examination of our rules suffices to see that the only instruction which might violate it is $unify\_value(y_n)$ in *Write* mode, with y_n a reference pointing outside the heap. If we call any such y_n (i.e. occurrence of

Y_n) *local*, we can enforce the constraint by using a special instruction *unify_local_value*, triggering a run-time check for locality, with

```
if  $code(p) = unify\_local\_value(l)$
    $\& \ mode = Write$
then  $succeed$
      if $local(deref(l))$
      then  $mk\_heap\_var(deref(l))$
            $trail(deref(l))$
      else  $h \leftarrow deref(l)$
            $h := h+$
```

where

$$local(l) \quad \equiv \quad unbound(l) \& NOT(l \in heap)$$

with a rule for *Read* mode of the same form as that for *unify_value*. The update $trail(deref(l))$ is of no import for Putting and Getting lemmas (and the Heap Variables lemma below), and thus can be regarded as a noop here; it will become significant in the next section.

The effect of *unify_local_value*, in terms of terms and substitutions constructed, is obviously the same as that of *unify_value* (up to substitution ordering), hence its usage in *put_code* function preserves the Putting and Getting lemmas. However, it preserves the Heap Variables Constraint as well, hence, given

Working Assumption. The *bind* update preserves the Heap Variables Constraint.

we have

Heap Variables Lemma. If *put_code* and *get_code* functions generate *unify_local_value* instead of *unify_value* for all occurrences of local variables, then the execution of *put_seq*,*get_seq* preserves the Heap Variables Constraint.

Proof is straightforward, up to the problem of recognizing occurrences of potentially local variables in Prolog terms. It will have to suffice to say here that this will include all occurrences of variables (within structures in body goals) for which the compiler cannot determine to be previously (in the given clause) allocated on the heap.

Example 10. In the clause $a(X) :- b(f(X))$ the second occurrence of X is local.

The Heap Variables Constraint will then be preserved under

Compiler Assumption 4. The assumptions of Heap Variables Lemma are satisfied.

The Working Assumption is temporary, since it will become a theorem when we discuss binding more carefully, in 4.2.2. below.

4.2 Prolog with Term Representation

Time has come to connect the Prolog model of section 3 to term representing algebras. The $STACK$ will have to be adapted, with the most notable change being the representation of substitutions, shared between environments and the $TRAIL$—a new stack used to record binding history.

4.2.1 The Stack and the Trail

The $STACK$ of section 3 will now take the form of a subalgebra of $DATAAREA$

$$(STACK; tos(b,e), bos; +, -; val) \quad b, e, ct \in STACK,$$

disjoint from $HEAP$ and $AREGS$, with the old decorating functions to be defined using $val, +, -$. The currently active finite part of the $STACK$ will be abbreviated as $stack \equiv \{l \in STACK \mid bos \leq l \leq tos(b,e)\}$ (finiteness follows from bos being the initial value of $tos(b,e)$).

The most notable difference from section 3 consists in the environments holding the variables $y_1, \ldots, y_n$, where n is recorded in the code for the last *call* executed, and accessible via $cp-$ (cf. Pure Prolog Theorem below). The information stored in the environment ('decorating functions') will be accessible by the function val applied to fixed offsets from e. We namely define functions ce, cp, y_i, tos

$$\begin{array}{rcl}
ce(l) & \equiv & l+1 \\
cp(l) & \equiv & l+2 \\
y_i & \equiv & e+2+i \quad 1 \leq i \leq stack\_offset(cp) \\
y_i(l) & \equiv & l+2+i \quad 1 \leq i \leq stack\_offset(val(cp(l))) \\
stack\_offset(l) & \equiv & n \quad \text{when } code(l-) = call(f,a,n) \\
tos(b,e) & \equiv & \begin{cases} \text{e+2+}stack\_offset(cp) & \text{if } b \leq e \\ \text{b} & \text{otherwise} \end{cases}
\end{array}$$

providing, so to speak, definitions for the abstract decorating functions of section 3—what used to be say $cp(e)$ is now reconstructible as $val(cp(e))$. [9] Current environment may then be depicted as

$\cdots$	ce	cp	y_1	$\cdots$	y_n

with e pointing below $ce \equiv ce(e)$.

The variable–renaming index is not needed any more, since different 'instances' of 'the same variable' now correspond two locations belonging to different places on the stack.

Any $y_i(l)$ for l in the ce-chain will be called a *stack variable* in the sequel. Environments will be created and discarded by the *environment-handling rules*, i.e. by executing homonymous instructions.

```
if $code(p) = allocate(n)$
then $e := tos(b,e)$
     $val(ce(tos(b,e))) := e$
     $val(cp(tos(b,e))) := cp$
     $succeed$
     seq $i = 1, \ldots, n$
       $mk\_unbound(y_i(tos(b,e)))$
     endseq
```

```
if $code(p) = deallocate$
then $e := val(ce(e))$
     $cp := val(cp(e))$
     $succeed$
```

Note that executing *allocate*, temporarily, includes initializing all new variables with *unbound*. This is a simplifying hypothesis to be dropped later.

Note also that, on *allocate*, we don't *create* a new object any more—the stack is now represented with pointers, and popped objects may be irretrievably lost, i.e. overwritten. The Hiding Lemma of 3.1. will now become significant, since its preservation will guarantee that an environment is still there whenever a choicepoint needs it.

[9] Note that we have suppressed the dependence on cp in definitions of y_i, tos.

The cutpoint *cutpt*(e) of section 3 has not been represented in the environment. Wishing to separate our concerns, we concentrate now on pure Prolog programs, i.e. those without any built-in predicates and constructs, including the *cut*. Once we have proved the correctness of the 'real' WAM wrt pure Prolog, ignoring all cutpoints, we shall, in 4.3.3, reintroduce the code and the data structures enabling correct execution of the *cut*, and patch up (the proof of) Pure Prolog Theorem below.

The decorating functions of a *choicepoint* will likewise be reconstructible by applying *val* to fixed offsets from b, with the addition of the current goal being represented by its arguments (the contents of argument registers), since its functor is accessible via $val(cp(b))-$, pictorially:

$\cdots$	x_n	$\cdots$	x_1	e	cp	b	p	tr	h

with b pointing at the top. To be completely formal, we have

$$\begin{array}{lll} h(l) \equiv l & tr(l) \equiv l - 1 & p(l) \equiv l - 2 \\ b(l) \equiv l - 3 & cp(l) \equiv l - 4 & e(l) \equiv l - 5 \\ x_i(l) \equiv l - 5 - i & hb \equiv val(h(b)) & \end{array}$$

Other layouts are of course possible, this is the original form of [Warren 83].

The indexing (choicepoint handling) instructions of section 2 (*try_me_else* | *try, retry_me_else* | *retry, trust_me* | *trust*) will now obtain an additional argument—the predicate arity n. Apart from that, only the *try* rule changes in significant way—we push not by creating but by overwriting.

```
if $code(p) = try\_me\_else(N, n) \mid try(C, n)$
then $b := new\_b$
        $val(b(new\_b)) := b$
        $store\_state\_in(new\_b, n)$
        $val(p(new\_b)) := N \mid val(p(new\_b)) := p+$
        $p := p+ \mid p := C$
where $new\_b = tos(b, e) + n + 6$
```

The reader could now easily reconstruct the present form of other indexing (and switching) rules, and of storing and fetching updates (or look them up in Appendix 4).

4.2.2 Binding, Trailing and Unbinding

Here we take a closer look at how the variables are bound and how these bindings are recorded, i.e. at the *bind* update. For the case of variable-to-variable binding, we need explicit assumptions about 'memory layout', in order to avoid the problem of 'dangling pointers', as the stack and the heap are going to grow and shrink at in general different rates.

We shall extend the ordering $<$, defined in terms of $+$ on the heap and the stack (and in terms of x on $AREGS$), assuming

WAM Assumption 3. $HEAP < STACK < AREGS$.

This assumption forces us to view (the active part of) *DATAAREA* as an ordered non-Archimedean structure (remember that *heap* and *stack* are finite at all times). A more realistic Archimedean *DATAAREA* would force us to consider resource limitations and garbage collection at this level, complicating every correctness statement with a qualification 'given sufficient resources'. We prefer to separate our concerns, and leave a treatment of resource limitations to a (possible) refinement of the present framework.

Given such an ordering, we further assume

WAM Assumption 4. If $unbound(l_1)$ & $unbound(l_2)$, then the effect of $bind(l_1, l_2)$ is to bind the higher location to the lower one.

Note that WAM Assumptions 3,4 imply our Working Assumption from 4.1.3. Further, we assume of *bind* to record variable bindings, to be undone in case of backtracking, on the *TRAIL*, i.e. on another algebra

$$(TRAIL, DATAAREA; tr, botr; +, -; ref'')$$

to be used as a stack. More precisely, we assume

WAM Assumption 5. Whenever *bind* binds a location l, it executes the update

```
$trail(l)$  ≡  $ref''(tr) := l$
               $tr := tr+$
```

The *TRAIL* essentially records the substitutions, stored previously explicitly in choicepoints. Since, whenever there is an alternative state of computation, $p(b)$ will point to an instruction of *retry, trust* family, the *backtrack* update must, except for passing control to $p(b)$, reconstruct the old substitution by undoing all bindings trailed after the current choicepoint was pushed, i.e.

```
$backtrack$  ≡  if $b = bos$
                then $stop := -1$
                else $p := val(p(b))$
                     seq $l = tr-, \ldots, tr(b)$
                        $mk\_unbound(ref''(l))$
                     endseq
```

By shrinking stack to b and *heap* to *hb*, the variables residing above these points will become irrelevant, and recording them on the *TRAIL* turns out to be spurious. An implementation could thus optimize by executing $trail(l)$ only under the condition $l \in heap$ & $l < hb$ or $l \in stack$ & $l < b$. This is however an optimization step without any semantical import, which we may disregard (simplifying thus somewhat the proof of Pure Prolog Theorem below).

WAM Assumptions 4,5 would become theorems had we *defined* the *bind* update as

```
if $NOT(unbound(l_2))$ or $l_1 > l_2$
then $l_1 \leftarrow l_2$
     $trail(l_1)$
elsif $l_2 > l_1$
then $l_2 \leftarrow l_1$
     $trail(l_2)$
```

Since we do not wish to exclude other, say occur-checking, implementations, we keep the assumptions. They ensure the

Stack Variables Property. Every stack variable l which doesn't point to an *AREG* containing a literal constant, points either to the heap or to a lower location of the stack, i.e if $l \in stack$ and $type(l) = Ref$ and $type(ref(l)) \neq Const$, then $ref(l) \in heap$ or $ref(l) \in stack$ & $ref(l) \leq l$.

Proof by inspection of rules—they all preserve the property, given WAM Assumptions 3,4. In case of $get\_value(x_j)$ note that x_j should have been put there by an instruction executed before the current environment was allocated.

4.2.3 Pure Prolog Theorem

At this point we may collect our assumption about the compiler in

Compiler Assumption 5. Assume functions *compile, unload* like in section 3, but now

$$\begin{array}{rcl} compile(H :- G_1, \ldots, G_n) & = & flatten(\, [\; allocate(r), get\_seq(H), \\ & & \quad call\_seq(G_1), \ldots call\_seq(G_n), \\ & & \quad deallocate, proceed \,]\,) \end{array}$$

where $call\_seq(g(s_1, \ldots, s_k)) \equiv flatten(\,[\,put\_seq(g(s_1, \ldots, s_k)), call(g, k, r)\,]\,)$, with $\{Y_1, \ldots, Y_r\}$ being all variables occurring in the clause.

Function *procdef* will now be typed by $ATOM \times ARITY \times PROGRAM \rightarrow CODEAREA$. Minding that the *call* instructions now have three arguments as above, it is a straightforward excercise to rewrite the rules *query success*, *proceed*, *call* of section 3 in this framework (cf. Appendix 4).

In the **initial state** with query $G_1, \ldots, G_n$, where all variables occurring in the query are $\{Y_1, \ldots, Y_r\}$, we shall have

$$unload(p) \quad = \quad flatten(\,[call\_seq(G_1), \ldots call\_seq(G_n), proceed\,]\,)$$

while initial environment, pointed at by e, will have all y_i initialized to *unbound*. Note that the initial environment makes it possible to introduce a sensible notion of *output substitution*.

At this point we can describe a mapping $\mathcal{F}$, mapping states of our current algebra to states of section 3 Prolog model, and (sequences of) our current rules to those of section 3. In view however of our limitation to non-STO unification only (cf. discussion in 4.1.1.), we have, for the sake of proof, to *constrain* the section 3 model by considering its abstract *unify* function as undefined in case unification is STO. This constraint is not meant as a modification of our model—it just expresses an attitude: we do not care what happens in that case, and we do not claim anything about it.

Setting up a correspondence we shall presently ignore cutpoints *ct, cutpt* (since they are irrelevant for pure Prolog) and variable indexes *vi*, (since their role of unique renaming is obviously taken over by offsets on the stack and the heap; if the reader cares about full detail, he can easily reintroduce them in our framework and extend the rules to handle the indexes like in section 3—they would never be used, except for reconstruction of a full section 3 state).

Since we have an exact correspondence of instructions and rules, a (partial) map of current *instruction sequences* to those of section 3.2 will simultaneously provide maps of code pointers (elements of *CODEAREA*) and of *rule sequences*, i.e. the rule component of the proof map $\mathcal{F}$. Instructions *goal* and *query success, allocate*, *deallocate*, *proceed* and all indexing and switching instructions will map to their homonyms. Further we have

$$\begin{array}{rcl} get\_seq(G) & \rightarrow & [\,unify(G)\,] \\ call\_seq(G) & \rightarrow & [\,call(G)\,] \end{array}$$

Marking all names of universes and functions of section 3 with an index 3, we have defined a partial function

$$codepointer : CODEAREA \rightarrow CODEAREA_3$$

which may be left undefined inside *get_sequences* and *call_sequences*. The rule component of the proof map $\mathcal{F}$ is thus defined as well.

The state component of $\mathcal{F}$ will then be defined inductively using the partial functions

$$\begin{array}{rcl} subst & : & TRAIL \longrightarrow SUBST \\ choicepoint & : & STACK \longrightarrow STATE_3 \\ env & : & STACK \longrightarrow ENV_3 \end{array}$$

and an auxiliary function

$$term : DATAAREA \times TRAIL \longrightarrow TERM$$

which can be defined as follows, taking an object of $STATE$ or ENV to be given by its s, p, cp, e, b or cp, e values, respectively.

$$\begin{array}{rcl} term(l, l_t) & & \text{yields the value } term(l) \text{ would take after having unwound the trail to } l_t \\ subst(l_t) & & \text{associates } mk\_var(ref''(l)) \text{ to } term(ref''(l), l_t),\ l < tr \\ choicepoint(lb) & = & \langle\ subst(val(tr(lb))), \\ & & \quad codepointer(val(p(lb))), \\ & & \quad codepointer(val(cp(lb))), \\ & & \quad env(val(e(lb))), \\ & & \quad choicepoint(val(b(lb)))\ \rangle \\ env(le) & = & \langle\ codepointer(val(cp(le))), env(val(ce(le)))\ \rangle \end{array}$$

understanding that a section 3.2 choicepoint or environment is given by its s, p, cp, e, b or cp, ce values, respectively, with inductive basis

$$choicepoint(bottom) = env(bottom) = bottom.$$

The $0-$ary functions of section 3 are then defined as

$$\begin{array}{rcl} s_3 & = & subst(tr) \\ p_3 & = & codepointer(p) \\ cp_3 & = & codepointer(cp) \\ e_3 & = & env(e) \\ b_3 & = & choicepoint(b) \end{array}$$

which establishes the state component of the proof map F. Although not required by the signature, it will be useful to verify maintenance of

$$act \quad = \quad act_3$$

(up to variable renaming) under execution of corresponding rule sequences, where

$$\begin{array}{rcl} act & = & g(term(x_1), \ldots, term(x_m)) \\ & & \text{when } \ code(cp-) = call(g, m, r) \end{array}$$

and act_3 was defined in Proof of 3.2. In fact, commutativity of $\mathcal{F}$ with rule execution now reduces to straightforward, though somewhat tedious, verification, relying on Putting and Getting Lemmas and (whenever stack gets popped), Heap Variables Lemma and the Stack Variables Property. Since $\mathcal{F}$ obviously preserves initial and stopping states (with *stop* value), we have

Proposition 4.1. The WAM version of 4.1. is, given WAM Assumptions 1–5 and Compiler Assumptions 1–5, correct and complete for pure Prolog programs wrt Prolog model of 3.2 constrained to non-STO unification.

Chaining all our propositions, we obtain

Pure Prolog Theorem. The WAM version of 4.1. is, given WAM Assumptions 1–5 and Compiler Assumptions 1–5, correct and complete for pure Prolog programs wrt Prolog trees constrained to non-STO unification.

4.2.4 Environment Trimming and the Last Call Optimization

Wishing to discard (i.e. recycle) stack space as soon as possible, we want any stack variable to get superfluous as soon as the *put_seq* for the goal containing its last occurrence in the clause is executed. A compiler could then be clever with variable numbering and trim the environment on the fly by generating a decreasing sequence of environment sizes (in third arguments of *call* instructions, cf. [Ait-Kaci 91]. According to [Warren 83], '... variables are arranged in their environment in such a way that they can be discarded as soon as they are no longer *needed*.' (our italics).

Let us then say, for now, that a variable, occurring before or in a body goal G_i of a clause $H :- G_1, \ldots, G_n,\ 1 \leq i < n$, is *needed* at G_i if it occurs after it as well, i.e. in some $G_l,\ l > i$.

(This definition is going to change in Section 3, as we optimize the design extending the instruction set. The form of other definitions below, relying on that of being needed, will however remain intact.)

Environment trimming will consist in $call\_seq(g(t_1, \ldots, t_m))$ ending in $call(g, m, r)$, where r need not be the environment size any more—it will suffice that all variables needed at or after corresponding body goal occurrence $g(t_1, \ldots, t_m)$ are among $\{Y1, \ldots, Yr\}$, which is our **Compiler Assumption 6**.

Correctness of environment trimming (i.e. preservation of Putting Lemma across indexing instructions, *call*, *allocate*) would be ensured by the following property.

Argument Registers Property. If Y_k is no more needed in a body goal occurrence G, then after executing $put\_seq(G)$, for any x_i affected by that *put_seq*, the computation of $term(x_i)$ does not meet y_k.

In view of the Heap Variables Lemma and Stack Variables Property, examination of the rules shows that the property could only be violated by $put\_value(y_n, x_j)$, with Y_n no more needed, and $ref(y_n) > e$. Corresponding occurences of Y_n in the clause are termed *unsafe* [Warren 83, Ait-Kaci 91]. The first of the two conditions amounts to Y_n occurring in an argument position of a body goal in which it is no more needed.

The second condition would be excluded had Y_n occurred in the clause head (y_n would then be set by the *get_seq* to point to a previously existing object, represented below e) or in a structure (y_n would be set to point to the *heap*, by a *unify* instruction). Thus the following definition.

An occurence of a variable Y_n in an argument position of a body goal, in which it is no more needed, is *unsafe* if Y_n does not previously occur in the clause head or in a structure.

Example 11. The second occurence of Y in the clause $has\_a(X) :- generate(X, Y), test(Y).$ is unsafe.

Since such an occurrence may nevertheless point below e, by having been previously bound, a run-time check would be appropriate, replacing $deref(y_n)$ by a new heap variable, but only if strictly necessary. The *compile* function should thus be modified to emit a new instruction,

$put\_unsafe\_value(y_n, x_j)$, including such a check, instead of $put\_value(y_n, x_j)$, for each unsafe occurrence of Y_n. The new instruction is 'executed' by the rule

```
if $code(p) = put\_unsafe\_value(y_n, x_j)$
then $succeed$
     if $deref(y_n) > e$
     then $mk\_heap\_var(deref(y_n))$
          $trail(deref(y_n))$
          $x_j \leftarrow mk\_ref(h)$
     else $x_j \leftarrow deref(y_n)$
```

Note that $deref(y_n) > e$ condition implies that it is also unbound.

We can thus correctly assume environment trimming, reestablishing the Argument Registers Property, given

Compiler Assumption 7. The *put_seq* function generates *put_unsafe_value* instead of *put_value* for any unsafe variable occurrence.

Since no variable is needed at the last goal, however, the environment will by then be trimmed down to continuation pointer. As far as variables are concerned, the environment could then be deallocated earlier, before the last call. In view of continuation pointer, however, replacing $call(g, a, 0)$, *deallocate*, *proceed* by *deallocate*, $call(g, a, 0)$, *proceed* would be wrong—the *cp* value restored from the environment by *deallocate* would be overwritten by *call*. The last *call* should then be replaced by a special instruction, acting just like $call(g, a, 0)$ but without touching *cp*. It is usually called $execute(g, a)$.

```
if $code(p) = execute(g, a)$
   & $is\_user\_defined(g, a)$
thenif $code(procdef(g, a, db)) = nil$
then $backtrack$
else $p := procdef(g, a, db)$
     $ct := b$
```

Note that appending *proceed* after *execute* would be quite spurious—p would never get to point to it anyway. For the incredulous, induction over the number of nested calls (noting that the basis means calling a fact, which *does* execute *proceed*) proves

Continuation Passing Lemma. Successful execution of *deallocate*, $execute(g, a)$ leaves p with the value $cp(e)$ had before execution.

Replacing $call(g, a, 0)$, *deallocate*, *proceed* with *deallocate*, $execute(g, a)$ is usually called *the last call optimization*. Together with indexing, it allows to execute some determinate tail recursive calls[10] to arbitrary depth on a bounded stack. The same effect, achieved here as an automatic run-time property of the WAM, is in some other contexts realized by sophisticated analysis and transformation of source code.

We may then safely assume

Compiler Assumption 8. The *compile* function may perform the last call optimization, i.e. replace the *call_seq* for the last body goal by its *execute_seq*, where

$$execute\_seq(g(t_1, \ldots, t_m)) = flatten([\,put\_seq(g(t_1, \ldots, t_m)), deallocate, execute(g, m)\,])$$

[10] A clause of form $p :-\ \ldots, p.$ is tail recursive if within $\ldots$ p is not called, directly or indirectly.

The discussion of this section may be summed up as

Trimming Lemma. Environment trimming and LCO preserve the Pure Prolog Theorem.

4.3 The WAM

In case of a fact or of a chain-rule, i.e. a void or a singleton body, no variable will ever be needed. However, we cannot (yet) say that *allocate* and $get\_seq(H)$ commute, or that *deallocate* is inverse to *allocate*, which would obviate the need to have an environment at all. Before that we have to introduce special instructions, enabling us to handle the nowhere needed, *temporary*, variables, without accessing an environment.

Temporary variables and related optimizations will be introduced in 4.3.1, 4.3.2, making our model worthy of the above title. The *cut* will be rediscussed in 4.3.3.

4.3.1 Temporary Variables

A variable occurring in a clause is *temporary* if it is not needed in any body goal. A variable which is not temporary is *permanent*.

This may seem to be a weird way of saying that a temporary variable 'does not occur in more than one goal in the body, counting the head of the clause as part of the first goal' [Warren 83]. In Example 11, for instance, X is temporary while Y is permanent. Note however that our definition is parameterized by the notion of being *needed*, which we are going to optimize in 4.3.2, making the above equivalent to Warren's full definition. [Ait-Kaci 91] has also attempted to derive Warren's original definition by optimizing the more naive notion above, but notions of environment trimming and unsafe variables are not adapted smoothly, what has later to be patched up by a notion of 'delayed trimming' (see the discussion in op.cit. section 5.9). We have the three definitions, of *temporary variables, environment trimming* and *unsafe variables*, coupled by the notion of being *needed* (and controlled by Argument Registers Property), so that optimization preserves correctness in a quasi automatic way.

The very notion of a temporary variable arises by optimization, intended to allow us not to *allocate* it in the environment. The Argument Registers Property tells us that permanent variables serve as communication channels between goals in the same body, leaving to temporary variables the role of mere descriptors of goal structure, akin to X_k's generated by term normal forms. The compiler will thus represent them likewise with $AREGS$. If we, for temporary variables, replace Y_n, y_n everywhere by fresh X_i, x_i, all the variable handling instructions (*get_value*, *put_value*, *unify_value*, *unify_local_value*) will work fine for them, given the Argument Registers Property, if the x_i's are properly initialized to objects on the *heap* or on the *stack*.

We shall then initialize each temporary variable on the fly, at its first occurrence, taking care that it gets the same value a permanent variable (initialized previously to *unbound* by *allocate*) would get in its place. We thus modify the *compile* function so as to emit $get\_variable(x_i, x_j)$, $unify\_variable(x_i)$, $put\_variable(x_i, x_j)$ for the first occurrence of a temporary variable X_i in a head argument, structure argument, body goal argument, respectively, which is our **Compiler Assumption 9.** Since temporary variables need (and should) not be trailed, new instructions will be 'executed' by the rules

```
if $code(p) = get\_variable(l, x_j)$
then $l \leftarrow x_j$
     $succeed$

if $code(p) = put\_variable(x_i, x_j)$
then $mk\_heap\_var(x_i)$
     $x_j \leftarrow mk\_ref(h)$
     $succeed$
```

Note that our old rulesfor *unify_variable* work fine for fresh x_i as well. By inspection of rules we get

Initialization Lemma. Instructions $get\_variable(l, x_j)$, $unify\_variable(l)$, $put\_variable(l, x_j)$ are equivalent (up to trailing) to, given $l > e$, initializing l to *unbound* and executing $get\_value(l, x_j)$, $unify\_local\_value(l)$, $put\_unsafe\_value(l, x_j)$ respectively.

It is usual to note that $get\_variable(x_i, x_i)$ and $put\_value(x_i, x_i)$ have the effect of *succeed*, permitting the compiler to be clever about numbering of temporary variables, confusing them with proper argument registers in order to minimize data movement ('peep-hole optimization', [Ait-Kaci 91]).

In case of a fact or a chain rule there will be no permanent variables, and *allocate* would just store *cp* on the stack. Now we can say that *allocate* would commute with *get_seq* and that *deallocate* would be its inverse, permitting us not to *allocate* at all, satisfying

Compiler Assumption 10. A fact or a chain-rule may be compiled to, respectively,

$$flatten(\,[get\_seq(Fact), proceed]\,)$$
$$flatten(\,[get\_seq(Head), pure\_execute\_sequence(Goal)]\,),$$

where *pure_execute_sequence* is like *execute_sequence*, but without *deallocate*.

4.3.2 Trading Heap for Stack or What Is Needed

By our definitions a temporary variable, first occurring as an argument of a body goal, would always occupy a fresh heap cell. Since the stack, in view of LCO and environment trimming (and determinacy detection), may retract much more often than the heap, trading in a persistent heap location for a volatile stack location may be considered as optimization.

A one goal variable first occurring as argument of a body goal may thus be better classified as permanent after all (excluding of course variables occurring in the last goal, if we are to preserve LCO). Since our treatment of permanent variables would however make it unsafe, nothing would be gained (and a stack location would be wasted) unless we reconsider that treatment.

A simple solution is not to make it unsafe—i.e. to let it live (be needed) a little longer by protecting it from being trimmed. A modified definition of being needed could adjust everything—-variable classification, notion of unsafe and environment trimming.

A variable occurring before or in a body goal G_i of a clause $H :- G_1, \ldots, G_n$, $1 \leq i < n$, is *needed* there at G_i if it occurs after it as well, i.e. in some G_l, $l > i$, or if its first occurence in the clause is an argument position of G_i.

With this definition of being needed, a one-goal variable which first occurs in an argument position of a body goal (not the last one), will be needed there, hence will not be unsafe, will not be trimmed till after the goal is called, and will be permanent. The Argument Registers Property is preserved by definition, a heap cell is traded in for a stack cell, and the definition of a temporary variable becomes equivalent to the classical one:

> A temporary variable is a variable that has its first occurrence in the head or in a structure or in the last goal, and that does not occur in more than one goal in the body, where the head of the clause is counted as part of the first goal. [Warren 83]

Example 12.[Ait-Kaci 91] In the clause $a :- b(X, X), c.$ the variable X is by the latest definition permanent.

One final step of optimization will bring our model to complete compliance with WAM compilation of clauses: it doesn't make sense (any more) to initialize a permanent variable to unbound at *allocate*, just in order to bind it to something else at its first occurrence. Permanent variables can be initialized on the fly, like temporary ones.

The *allocate* rule will thus lose its initialization update. The cases of a permanent variable occurring first in the head or in a structure are handled correctly (we have verified that already) by the existing *get_variable* and *unify_variable* rules. An attentive reader will have noticed that they bind a permanent variable without trailing it, but this has no semantical consequences (cf. discussion in 4.2.2). The only consequence would be to our proof of Pure Prolog Theorem, where the definition of *subst* should now take into account all permanent variables, whether trailed or not.

A permanent variable first occurring in an argument position of a body goal by (the latest) definition cannot be unsafe there, so we still need an instruction equivalent to $\mathit{put\_value}(y_n, x_j)$ with $\mathit{unbound}(y_n)$. It is usually called $\mathit{put\_variable}(y_n, x_j)$, enabling us to formulate our **Compiler Assumption 11** by extending Compiler Assumption 9 to permanent variables.

```
if   $\mathit{code}(p) = \mathit{put\_variable}(y_n, x_j)$
then $\mathit{mk\_unbound}(y_n)$
     $x_j \leftarrow \mathit{mk\_ref}(y_n)$
     $\mathit{succeed}$
```

The instruction $\mathit{put\_variable}(x_i, x_j)$ is now reduced to the role of initializing those variables which first occur in an argument position of the last goal. The above discussion may then be summed up as

Classification Lemma. Warren's classification of variables and on_the_fly initialization preserve the Pure Prolog Theorem.

We have arrived at the model expressing all aspects of [Warren 83], which we then dare to call *the full WAM*. For the record, we might list the final code for the clauses of examples 10,11,12.

$a(X) :- b(f(X)).$	$\mathit{has\_a}(X) :-$ $\mathit{generate}(X, Y), \mathit{test}(Y).$	$a :- b(X, X), c.$
$\mathit{get\_variable}(x_2, x_1)$	$\mathit{allocate}(1)$	$\mathit{allocate}(1)$
$\mathit{put\_structure}(f, 1, x_1)$	$\mathit{put\_variable}(y_1, x_2)$	$\mathit{put\_variable}(y_1, x_1)$
$\mathit{unify\_local\_value}(x_2)$	$\mathit{call}(\mathit{generate}, 2, 1)$	$\mathit{put\_value}(y_1, x_2)$
$\mathit{execute}(b, 1)$	$\mathit{put\_unsafe\_value}(y_1, x_1)$	$\mathit{call}(b, 2, 1)$
	$\mathit{deallocate}$	$\mathit{deallocate}$
	$\mathit{execute}(\mathit{test}, 1)$	$\mathit{execute}(c, 0)$

4.3.3 The Cut

It would be straightforward to enable the WAM to execute the *cut* correctly, by imitating the treatment of section 3. We shall, like in section 3, maintain a 0-ary $ct \in \mathit{STATE}$, to be loaded from b on *call* | *execute*, and restored on backtracking. We could further have a function *cutpt* associating a *STATE* with every *ENV*, to be loaded from *ct* on *allocate*, executing *call*(!) by resetting b to $\mathit{cutpt}(e)$. It is a straightforward excercise to patch up the proof of Pure Prolog Theorem for such an extension, extending thereby the proof of correctness and completeness to Prolog with *cut*.

It would however be wasteful to *allocate* a *cutpt* in every environment, regardless of whether a *cut* could be executed within the corresponding clause body at all. A usual decision [Ait-Kaci 91] is to *allocate* a *cutpt* only when it could be needed, i.e. when the corresponding body *could* contain a *cut*[11].

A *neck_cut*, i.e. one which is the first goal of a body, does not need a cutpoint stored in the environment—*ct* certainly holds the correct cutpoint (since indexing and *allocate* don't touch it). If the compiler emits the appropriate instruction, a *neck_cut* would be correctly executed by the rule

```
if $code(p) = neck\_cut$
then $b := ct$
     $succeed$
```

A cutpoint should then be allocated in the environment only in case of a *deep cut* occurrence, i.e. one which is not the first goal in the body. In order to preserve uniformity of data representation, it is usual to allocate an extra permanent variable to hold the cutpoint. The compiler should then emit a special instruction, *save*(y_i), immediately after *allocate*, to store the cutpoint in the *cutpoint variable* y_i, and, in place of the *cut*, a special instruction *cut*(y_i), which thus, by its form, knows where the proper cutpoint is to be found.

```
if $code(p) = save(y_i)$          if $code(p) = cut(y_i)$
then $val(y_i) := ct$             then $b := val(y_i)$
     $succeed$                         $succeed$
```

Note that a cutpoint variable is needed at every goal in or after which it occurs, even if it is the last one—a *cut* occurring as the last goal of a clause precludes LCO.

Interpreting the cutpoint variable, when it exists, as *cutpt*(e)[12], the proof of Pure Prolog Theorem extends to pure Prolog with *cut*, given

Compiler Assumption 12. The *compile* function emits a *neck_cut* instruction for a neck cut in a clause body, while, for a deep cut, the instruction *cut*(y_i) is emitted, where y_i is a specially allocated cutpoint variable. The cutpoint variable y_i is permanent, needed before and at every *cut*(y_i). In code for any clause for which a cut variable is allocated, *allocate* is immediately followed by the corresponding *save*(y_i).

The Pure Prolog Theorem and subsequent preservation results yield

Main Theorem. The full WAM is, given WAM Assumptions 1–5 and Compiler Assumptions 1–12, correct and complete wrt Prolog trees constrained to non-STO unification.

Corollary. The full WAM is, given WAM Assumptions 1–5 and Compiler Assumptions 1–11, correct, for pure Prolog programs, wrt SLD-resolution constrained to non-STO unification.

The Corollary follows immediately, given the Prolog Tree Theorem of [Boerger,Rosenzweig 93]

[11] In view of the *metacall* construct of full Prolog, i.e. possibility of a variable in place of a body goal, every such variable could be at run time instantiated to a *cut*.

[12] When a cut variable is not allocated, *cutpt*(e) is not used in the model of section 3 anyway, except for a neck cut, for which *ct* suffices.

References

[Ait-Kaci 91] Aït-Kaci, H. *Warren's Abstract Machine. A Tutorial Reconstruction.* MIT Press 1991.

[Apt 90] Apt, C. *Logic Programing.* In: J.van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier, Vol.B, 1990, 493-574

[Beierle,Boerger 92] Beierle, C. & Börger, E. *Correctness proof for the WAM with types.* In: Computer Science Logic 91. (Eds. E.Börger, G.Jäger, H.Kleine-Büning,M.Richter), Springer LNCS 626 , 1992, pp. 15–34

[Boerger 90a] Börger, E. *A logical operational semantics of full Prolog: Part 1. Selection core and control.* In: CSL '89, 3rd Workshop on Computer Science Logic (Eds. E.Börger, H.Kleine-Büning,M.Richter), Springer LNCS 440, 1990, pp. 36-64

[Boerger 90b] Börger, E. *A logical operational semantics of full Prolog: Part 2. Built-in predicates for database manipulations.* In: MFCS '90 (Ed. B.Rovan, Springer LNCS 452, 1990, pp. 1-14

[Boerger,Rosenzweig 91a] Börger, E. & Rosenzweig D. *From Prolog Algebras Towards WAM—A Mathematical Study of Implementation.* In: CSL '90, 4th Workshop on Computer Science Logic (Eds. E.Börger, H.Kleine-Büning,M.Richter,W.Schönfeld), Springer LNCS 533, 1991, pp. 31-66

[Boerger,Rosenzweig 91b] Börger, E. & Rosenzweig D. *An Analysis of Prolog Database Views and Their Uniform Implementation* CSE-TR-88-91, University of Michigan, Ann Arbor, Michigan, 1991.

[Boerger,Rosenzweig 91c] Börger, E. & Rosenzweig D. *WAM Algebras—A Mathematical Study of Implementation, Part 2.* In: Logic Programming (Ed. A. Voronkov). Springer LNCS 592, 1991, pp.35-54

[Boerger,Rosenzweig 93] Börger, E. & Rosenzweig D. *A Mathematical Definition of Full Prolog*, to appear in The Science of Computer Programming (also appeared as Technical Report TR–33/92, Dipartimento di Informatica, Università di Pisa 1992)

[Glavan,Rosenzweig 93] Glavan, P. & Rosenzweig, D. *Communicating Evolving Algebras*, in: Computer Science Logic, Selected Papers from CSL'92, (E.Börger, G.Jäger, H. Kleine Büning, S.Martini, M.M.Richter eds.) Springer LNCS 702, pp. 182–215

[Börger,Salamone 94] E.Börger,R.Salamone, *CLAM Specification for Provably Correct Compilation of CLP(R) Programs.* In: *Specification and Validation Methods for Programming Languages and Systems* (E.Börger, Ed.), Oxford University Press, 1994 (to appear)

[Gurevich 88] Gurevich,Y. *Logic and the challenge of computer science.* in: E.Börger (Ed.), Trends in Theoretical Computer Science. Computer Science Press, Rockville MA 1988, pp.1-57

[Gurevich 91] Gurevich,Y. *Evolving Algebras. A Tutorial Introduction.* in: Bulletin of the European Association for Theoretical Computer Science, no.43, February 1991, pp. 264-284

[ISO WG17 93] *Prolog. Part 1, General core. Committee Draft* ISO/IEC JTCI SC22 WG17 N.110, 1993.

[Lindholm,O'Keefe 87] Lindholm, T.G. & O'Keefe, R.A. *Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code.* In: Proceedings of the Fourth International Conference on Logic Programming, pp. 21-39

[Lloyd 84] Lloyd, J.W. *Foundations of Logic Programming*, Springer, Berlin-Heidelberg-New York 1984.

[Russinoff 92] Rusinoff, M.D. *A Verified Prolog Compiler for the Warren Abstract Machine*, Journal of Logic Programming 13(1992), 367–412.

[Warren 83] Warren, D.H.D. *An Abstract Prolog Instruction Set*, Technical Note 309, Artificial Intelligence Center, SRI International 1983.

[Wirsing 90] Wirsing, M. *Algebraic Specification*, in J. van Leeuwen (Ed.): Handbook of Theoretical Computer Science B, Elsevier 1990, pp. 675–788.

Appendix 1. Rules for the Prolog Tree Model

```
if $all\_done$
then $stop := 1$

if $is\_user\_defined(act)$
    & $mode = Call$
then
  extend $NODE$ by $temp_1, \ldots, temp_n$
  with
    $father(temp_i) := currnode$
    $cll(temp_i) := nth(procdef(act, db), i)$
    $cands := [temp_1, \ldots, temp_n]$
  endextend
  $mode := Select$
where $n = length(procdef(act, db))$

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

if $act = true$
then $succeed$
```

```
if $goal = [\,]$
then $decglseq := rest(decglseq)$

if $is\_user\_defined(act)$
    & $mode = Select$
thenif $cands = [\,]$
then $backtrack$
elsif $unify = nil$
then $cands := rest(cands)$
else $currnode := fst(cands)$
       $decglseq(fst(cands)) :=$
          $subres([\,\langle bdy(clause), father\rangle$
            $\mid cont\,], unify)$
       $s(fst(cands)) := s \circ unify$
       $cands := rest(cands)$
       $mode := Call$
       $vi := vi + 1$
where
    $clause =$
      $rename(clause(cll(fst(cands))), vi)$
    $unify = unify(act, hd(clause))$

if $act = fail$
then $backtrack$
```

under the abbreviations

```
$father \equiv father(currnode)$
$cands \equiv cands(currnode)$
$s \equiv s(currnode)$
$decglseq \equiv decglseq(currnode)$
$cont \equiv [\,\langle\, rest(goal), cutpt\,\rangle \mid tail(decglseq)\,]$

$backtrack \equiv$ if $father = root$
                 then $stop := -1$
                 else $currnode := father$
                        $mode := Select$

$goal \equiv fst(fst(decglseq))$
$act \equiv fst(goal)$
$cutpt \equiv snd(fst(decglseq))$
$succeed \equiv decglseq := cont$

$all\_done \equiv decglseq = [\,]$
```

Appendix 2. Rules for Compiled Predicate Structure

```
if $all\_done$
then $stop := 1$

if  $p = start$
    & $is\_user\_defined(act)$
thenif $code(procdef(act, db)) = nil$
then $backtrack$
else  $p := procdef(act, db)$
        $ct := b$

if $act = !$
then  $b := cutpt$
         $succeed$

if $act = true$
then $succeed$
```

```
if $goal = [\,]$
then $decglseq := rest(decglseq)$

if  $code(p) \in CLAUSE$
    & $is\_user\_defined(act)$
thenif $unify = nil$
then $backtrack$
else
  $decglseq :=$
     $subres(\ \ [\langle bdy(clause), ct\rangle \mid cont],$
              $unify\ )$
  $s := s \circ unify$
  $p := start$
  $vi := vi + 1$
where
  $clause = rename(code(p), vi)$
  $unify = unify(act, hd(clause))$

if $act = fail$
then $backtrack$
```

Indexing and switching rules

```
if $code(p) = try\_me\_else(N) \mid try(C)$
then push $temp$ with
  $store\_state\_in(temp)$
  $p(temp) := N \mid p(temp) := p+$
endpush
$p := p+ \mid p := C$

if $code(p) = switch\_on\_term(i, Lv, Lc, Ll, Ls)$
thenif  $is\_var(x_i)$
then $p := Lv$
elsif $is\_const(x_i))$
then $p := Lc$
elsif $is\_list(x_i)$
then $p := Ll$
thenif $is\_struct(x_i)$
then $p := Ls$
```

```
if $code(p) = retry\_me\_else(N) \mid retry(C)$
then  $fetch\_state\_from(b)$
         $restore\_cutpoint$
         $p(b) := N \mid p(b) := p+$
         $p := p+ \mid p := C$

if $code(p) = trust\_me \mid trust(C)$
then  $fetch\_state\_from(b)$
         $restore\_cutpoint$
         $b := b(b)$
         $p := p+ \mid p := C$

if $code(p) = switch\_on\_constant(i, N, T)$
then $p := hash_c(T, N, x_i)$

if $code(p) = switch\_on\_structure(i, N, T)$
then $p := hash_s(T, N, funct(x_i), arity(x_i))$
```

under all abbreviations of Appendix 1., but for

```
$backtrack$  $\equiv$  if $b = bottom$
                       then$stop := -1$
                       else $p := p(b)$

$x_i$  $\equiv$  $arg(act, i)$

$store\_state\_in(t)$  $\equiv$  $decglseq(t) := decglseq$
                                 $s(t) := s$

$fetch\_state\_from(t)$  $\equiv$  $decglseq := decglseq(t)$
                                   $s := s(t)$
```

Appendix 3. Rules for Compiled Clause Structure

```
if $all\_done$
then $stop := 1$

if $code(p) = allocate$
then
  allocate $temp$ with
    $cp(temp) := cp$
    $vi(temp) := vi$
    $cutpt(temp) := ct$
  endalloc
  $succeed$

if  $code(p) = call(G)$
    & $is\_user\_defined(G)$
thenif $code(procdef(act, db)) = nil$
then $backtrack$
else  $p := procdef(act, db)$
      $cp := p+$
      $ct := b$

if $code(p) = call(true)$
then $succeed$

if $code(p) = call(!)$
then  $b := cutpt$
      $succeed$
```

```
if  $code(p) = proceed$
    & $NOT(all\_done)$
then $p := cp$

if $code(p) = deallocate$
then  $e := ce(e)$
      $cp := cp(e)$
      $succeed$

if $code(p) = unify(H)$
thenif $unify = nil$
then $backtrack$
else  $s := s \circ unify$
      $vi := vi + 1$
      $succeed$
where
  $unify = unify(act, rename(H, vi))$

if $code(p) = call(fail)$
then $backtrack$
```

with indexing and switching rules ¿from Appendix 2, as well as all abbreviations but for

```
$all\_done$  ≡  $code(p) = proceed$
                & $code(cp) = proceed$

$act$  ≡  $subres(rename(G, vi(e)), s)$
          where $code(cp-) = call(G)$

$store\_state\_in(t)$  ≡  $cp(t) := cp$
                          $s(t) := s$
                          $e(t) := e$

$fetch\_state\_from(t)$  ≡  $cp := cp(t)$
                            $s := s(t)$
                            $e := e(t)$

allocate $t$ with      ≡   extend $ENV$ by $t$ with
  $updates(t)$               $e := t$
endalloc                     $ce(t) := e$
                             $t- := tos(b, e)$
                             $updates(t)$
                           endextend
```

Appendix 4. Rules for the WAM

```
if $all\_done$
then $stop := 1$
```

```
if  $code(p) = proceed$
    & $NOT(all\_done)$
then $p := cp$
```

```
if $code(p) = allocate(n)$
then  $e := tos(b, e)$
      $val(ce(tos(b, e))) := e$
      $val(cp(tos(b, e))) := cp$
      $succeed$
```

```
if $code(p) = deallocate$
then  $e := val(ce(e))$
      $cp := val(cp(e))$
      $succeed$
```

```
if  $code(p) = call(g, a, r)$
    & $is\_user\_defined(g, a)$
thenif $code(procdef(g, a, db)) = nil$
then $backtrack$
else  $p := procdef(g, a, db)$
      $cp := p+$
      $ct := b$
```

```
if  $code(p) = execute(g, a)$
    & $is\_user\_defined(g, a)$
thenif $code(procdef(g, a, db)) = nil$
then $backtrack$
else  $p := procdef(g, a, db)$
      $ct := b$
```

Putting

```
if $code(p) = put\_variable(y_n, x_j)$
then  $mk\_unbound(y_n)$
      $x_j \leftarrow mk\_ref(y_n)$
      $succeed$
```

```
if $code(p) = put\_variable(x_i, x_j)$
then  $mk\_heap\_var(x_i)$
      $x_j \leftarrow mk\_ref(h)$
      $succeed$
```

```
if $code(p) = put\_value(l, x_j)$
then  $x_j \leftarrow l$
      $succeed$
```

```
if $code(p) = put\_unsafe\_value(y_n, x_j)$
then
  $succeed$
  if $deref(y_n) > e$
  then
    $mk\_heap\_var(deref(y_n))$
    $trail(deref(y_n))$
    $x_j \leftarrow mk\_ref(h))$
  else
    $x_j \leftarrow deref(y_n)$
```

```
if $code(p) = put\_constant(c, x_j)$
then  $x_j \leftarrow \langle Const, c \rangle$
      $succeed$
```

```
if $code(p) = put\_list(x_i)$
then  $h \leftarrow \langle List, h+ \rangle$
      $x_i \leftarrow \langle List, h+ \rangle$
      $h := h+$
      $mode := Write$
      $succeed$
```

```
if $code(p) = put\_structure(f, a, x_i)$
then  $h \leftarrow \langle Struct, h+ \rangle$
      $h+ \leftarrow \langle Funct, \langle f, a \rangle \rangle$
      $x_i \leftarrow \langle Struct, h+ \rangle$
      $h := h++$
      $mode := Write$
      $succeed$
```

Getting

```
if $code(p) = get\_variable(l, x_j)$
then $l \leftarrow x_j$
     $succeed$
```

```
if $code(p) = get\_value(l, x_j)$
then $unify(l, x_j)$
     $succeed$
```

```
if $code(p) = get\_list(x_i)$
thenif $type(deref(x_i)) = Ref$
then $deref(x_i) \leftarrow \langle List, h+ \rangle$
     $trail(deref(x_i))$
     $h \leftarrow \langle List, h+ \rangle$
     $h := h+$
     $mode := Write$
     $succeed$
elsif $type(deref(x_i)) = List$
then $str := ref(deref(x_i))$
     $mode := Read$
     $succeed$
else $backtrack$
```

```
if $code(p) = get\_constant(c, x_i)$
thenif $type(deref(x_i)) = Ref$
then $deref(x_i) \leftarrow \langle Const, c \rangle$
     $trail(deref(x_i))$
     $succeed$
elsif $type(deref(x_i)) = Const$
      $\& \ ref(deref(x_i)) = c$
then $succeed$
else $backtrack$
```

```
if $code(p) = get\_structure(f, a, x_i)$
thenif $type(deref(x_i)) = Ref$
then $deref(x_i) \leftarrow \langle Struct, h+ \rangle$
     $trail(deref(x_i))$
     $h \leftarrow \langle Struct, h+ \rangle$
     $h+ \leftarrow \langle Funct, \langle f, a, \rangle \rangle$
     $h := h++$
     $mode := Write$
     $succeed$
elsif $type(deref(x_i)) = Struct$
      $\& \ ref(ref(deref(x_i))) = \langle f, a \rangle$
then $str := ref(deref(x_i))+$
     $mode := Read$
     $succeed$
else $backtrack$
```

Unifying

```
if $code(p) = unify\_variable(l)$
then $succeed$
     if $mode = Read$
     then $l \leftarrow str$
          $str := str+$
     else $mk\_heap\_var(l)$
```

```
if $code(p) = unify\_local\_value(l)$
then $succeed$
     if $mode = Read$
     then $unify(l, str)$
          $str := str+$
     elsif $local(deref(l))$
     then $mk\_heap\_var(deref(l))$
          $trail(deref(l))$
     else $h \leftarrow deref(l)$
          $h := h+$
```

```
if $code(p) = unify\_value(l)$
then $succeed$
     if $mode = Read$
     then $unify(l, str)$
          $str := str+$
     else $h \leftarrow l$
          $h := h+$
```

```
if $code(p) = unify\_constant(l)$
if $mode = Read$
then ( if $type(deref(str)) = Ref$
       then $deref(str) \leftarrow \langle Const, c \rangle$
            $trail(deref(str))$
            $str := str+$
            $succeed$
       elsif $type(deref(str)) = Const$
             $\& \ ref(deref(str)) = c$
       then $str := str+$
            $succeed$
       else $backtrack$ )
else $h \leftarrow \langle Const, c \rangle$
     $h := h+$
     $succeed$
```

Indexing and Switching

```
if $code(p) = try\_me\_else(N, n) \mid try(C, n)$
then $b := new\_b$
        $val(b(new\_b)) := b$
        $store\_state\_in(new\_b, n)$
        $val(p(new\_b)) := N \mid p+$
        $p := p+ \mid C$
where $new\_b = tos(b, e) + n + 6$
```

```
if $code(p) = switch\_on\_term(i, Lv, Lc, Ll, Ls)$
thenif $type(deref(x_i)) = Ref$
then $p := Lv$
elsif $type(deref(x_i)) = Const$
then $p := Lc$
elsif $type(deref(x_i)) = List$
then $p := Ll$
elsif $type(deref(x_i)) = Struct$
then $p := Ls$
```

```
if $code(p) = retry\_me\_else(N, n)$
                    $\mid retry(C, n)$
then $fetch\_state\_from(b, n)$
        $restore\_cutpoint$
        $val(p(b)) := N \mid p+$
        $p := p+ \mid C$
```

```
if $code(p) = trust\_me(n) \mid trust(C, n)$
then $fetch\_state\_from(b, n)$
        $restore\_cutpoint$
        $b := val(b(b))$
        $p := p+ \mid C$
```

```
if $code(p) = switch\_on\_constant(i, N, T)$
then $p := hash_c(T, N, ref(deref(x_i)))$
```

```
if $code(p) = switch\_on\_structure(i, N, T)$
then $p := hash_s(T, N, ref(ref(deref(x_i))))$
```

where

```
$store\_state\_in(t, n)$    $\equiv$    seq $i = 1, \ldots, n$
                                  $val(x_i(t)) := x_i$
                                endseq
                                $val(e(t)) := e$
                                $val(cp(t)) := cp$
                                $val(tr(t)) := tr$
                                $val(h(t)) := h$
```

```
$fetch\_state\_from(t, n)$    $\equiv$    seq $i = 1, \ldots, n$
                                    $x_i := val(x_i(t))$
                                  endseq
                                  $e := val(e(t))$
                                  $cp := val(cp(t))$
                                  $tr := val(tr(t))$
                                  $h := val(h(t))$
```

Built-in Constructs

```
if $code(p) = call(true) \mid execute(true)$
then $p := p+ \mid cp$
```

```
if $code(p) = save(y_i)$
then $val(y_i) := ct$
        $succeed$
```

```
if $code(p) = neck\_cut$
then $b := ct$
        $succeed$
```

```
if $code(p) = call(fail) \mid execute(fail)$
then $backtrack$
```

```
if $code(p) = cut(y_i)$
then $b := val(y_i)$
        $succeed$
```

where

```
$backtrack$    $\equiv$    if $b = bos$
                    then $stop := -1$
                    else $p := val(p(b))$
                            seq $l = tr-, \ldots, tr(b)$
                              $mk\_unbound(ref''(l))$
                            endseq
```