A Verified Prolog Compiler for the Warren Abstract Machine

David M. Russinoff

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, TX 78759
(512) 338-3583

Abstract

We extend the theory of Prolog to provide a framework for the study of Prolog compilation technology. For this purpose, we first demonstrate the semantic equivalence of two Prolog interpreters: a conventional SLD-refutation procedure and one that employs Warren's "last call" optimization. Next, we formally define the Warren Abstract Machine (WAM) and its instruction set and present a Prolog compiler for the WAM. Finally, we prove that the WAM execution of a compiled Prolog program produces the same result as the interpretation of its source.

Contents

1 Introduction

One reason for the dominance of Prolog [6] as the canonical language of logic programming is its susceptibility to efficient implementation. D. H. D. Warren's contribution to the art of Prolog compilation [10] represents a major breakthrough in this area. Most high-performance Prolog systems are based on the Warren Abstract Machine (WAM), and considerable current research is devoted to various modifications and extensions of the WAM. These include investigations of its applicability to areas beyond the realm of logic programming, such as automatic theorem proving [5] and expert systems technology [4].

Prolog inherits a well established mathematical foundation from its logical roots — its semantics are clearly defined and thoroughly understood [8]. In contrast, advances in the implementation of Prolog are generally results of engineering efforts rather than mathematical investigations. Consequently, there is no rigorous theory of Prolog compilation. In particular, the principles underlying the design of the WAM have not been adequately explicated.

Just as the semantic development of Prolog has benefited from its grounding in logic, the advancement of Prolog technology could be facilitated by the establishment of a theoretical framework for compilation. In the absence of such a framework, our confidence in the correctness of the WAM must be largely based on successful testing. Any new implementation or functional extension, therefore, requires further testing. As these extensions continue to grow in complexity, a more convincing verification method becomes increasingly desirable.

The purpose of this paper is to extend the theory of Prolog to provide a mathematical basis for Prolog compilation technology, including a specification of the WAM and a proof of its correctness. In section 2, we begin with a review of the syntax of Prolog and define its semantics in terms of an SLD-refutation procedure [2]. We then present a modified Prolog interpreter, incorporating Warren's *last call optimization* technique, and prove that Prolog semantics are preserved by this modification.

In Section 3, we present an abstract definition of the WAM and its instruction set. This definition is intended to capture the central features of Warren's design [11], including the tail recursion and last call optimizations. However, we are interested only in formalizing the essential WAM and establishing its correctness, not in explaining why it is optimized precisely as it is. Thus, we introduce a number of minor modifications, designed to simplify our model of the WAM and thereby facilitate its analysis. None of these modifications, however, significantly affects either its functionality or the validity of our results:

1. Several specialized WAM instructions, which were added by Warren to allow various local performance optimizations, are eliminated. Thus, the instructions `GET-LIST`, `PUT-LIST`, `GET-NIL`, `PUT-NIL`, `UNIFY-NIL`, and `UNIFY-VOID` are subsumed by the more general `GET-STRUCTURE`, `PUT-STRUCTURE`, `GET-CONSTANT`, `PUT-CONSTANT`, `UNIFY-CONSTANT`, and `UNIFY-VARIABLE`, respectively.

2. *Environment trimming*, a technique for minimizing the size of the local stack, is eliminated. This results in simplification of the `CALL` instruction as well as the definition of *permanent variable*. It also allows the environment register to point to the top, rather than the bottom, of the current environment. The backtrack register similarly points to the top of the most recent choice point, thus allowing the top of the stack to be computed as the maximum of the values of these two registers.

3. The process of allocating a structure on the heap is modified so that the arguments of a new structure are initialized as unbound variables. This eliminates *write mode* and `UNIFY-LOCAL-VALUE` and simplifies the other unification instructions.

4. Our most radical departure is the elimination of indexing. Choice points are allocated and maintained by means of the `TRY-ME-ELSE`, `RETRY-ME-ELSE`, and `TRUST-ME-ELSE-FAIL` instructions; the other indexing instructions, `TRY`, `RETRY`, `TRUST`, and the `SWITCH-` instructions, are not used.

Section 4 discusses Prolog compilation on the WAM and the execution of compiled programs. We present a complete compiler for pure Prolog (although in the interest of brevity, we ignore various opportunities for local optimizations), and prove the equivalence between the execution of a compiled Prolog program and the interpretation of its source. That is, we show that the interpretive semantics of Prolog, as defined in Section 2, are implemented by the WAM emulator and compiler of Sections 3 and 4. The proof involves a correspondence between WAM states and the states of a Prolog interpreter.

While this paper is intended to be technically self-contained, it will lack motivation for anyone who is unfamiliar the subject. The uninitiated reader is advised to consult other sources on both the interpretive semantics of Prolog [6,8] and the WAM [1,7,11].

Our theory, which is influenced by the work of Boyer and Moore [3], is based solely on naive set theory and relies heavily on mathematical induction and recursive function definition. We assume the existence of a universal set $\mathbf{U}$, which is the disjoint union of the set $\mathbf{N} = \{0, 1, 2, \ldots\}$ of natural numbers and various other sets, which will be introduced as we proceed:

$$\mathbf{U} = \mathbf{N} \cup \mathbf{F} \cup \mathbf{P} \cup \mathbf{K} \cup \mathbf{V} \cup \mathbf{R} \cup \mathbf{A} \cup \mathbf{S} \cup \mathbf{H} \cup \mathbf{T} \cup \mathbf{C} \cup \mathbf{O} \cup \mathbf{X} \cup \mathbf{Y}.$$

An *expression* is defined recursively to be an element of $\mathbf{U}$ or an ordered pair of expressions. The *complexity* of an expression E is defined as follows: if $E \in \mathbf{U}$, then $complexity(E) = 1$, and if $E = (E_1, E_2) \notin \mathbf{U}$, then $complexity(E) = complexity(E_1) + complexity(E_2)$.

In order to facilitate the definitions of various complicated recursive functions, including those that comprise the compiler, we adopt a pseudocode notation (see Figures 1–7). This notation should be self-explanatory. Moreover, it should be apparent that each of these definitions could be rewritten (at the expense of readability) in a more standard mathematical notation, e.g., as a formula of predicate logic.

In dealing with ordered pairs, we borrow the function names *car* and *cdr* from LISP: if z is the ordered pair (x, y), then $x = car(z)$ and $y = cdr(z)$. We also borrow from LISP the symbol `NIL`, which we shall assume to be an element of the set $\mathbf{K}$ (and hence an element of $\mathbf{U}$). For convenience, we define $car(\mathtt{NIL})$ and $cdr(\mathtt{NIL})$ to be `NIL`. A *list* is defined recursively to be either the symbol `NIL` or an ordered pair z such that $cdr(z)$ is a list. The list `NIL` will also be represented as $\langle\rangle$, and a non-`NIL` list z will be represented as $\langle a_n, \ldots, a_1\rangle$, where $a_n = car(z)$ and $\langle a_{n-1}, \ldots, a_1\rangle$ represents $cdr(z)$. The *length* of a list $\langle a_n, \ldots, a_1\rangle$ is defined to be the natural number n, and $a_n, \ldots, a_1$ are its *members*. Thus, `NIL` is the unique list of length 0, and $\langle x, y, z\rangle = (x, (y, (z, \mathtt{NIL})))$ is a list of length 3 with members x, y, and z.

A list y is a *tail* of a list x if either $y = x$ or y is a tail of $cdr(x)$. y is a *sublist* of x if a) $y = \texttt{NIL}$, b) $x = (u, v)$ and y is a sublist of v, or c) $x = (u, v)$ and $y = (u, w)$, where w is a sublist of v.

Let $x = \langle a_n, \ldots, a_1 \rangle$ be a list. In accordance with the standard LISP definition, we define $member(a_k, x)$, for each k, $n \geq k \geq 1$, to be the tail $\langle a_k, a_{k-1}, \ldots, a_1 \rangle$ if $a_i \neq a_k$ for $n \geq i > k$, while if b is not any of the a_i, then $member(b, x)$ is defined to be NIL. If $y = \langle b_m, \ldots, b_1 \rangle$ is a second list, then $append(x, y) = \langle a_n, \ldots, a_1, b_m, \ldots, b_1 \rangle$. If $z = append(x, y)$, then we may also write $x = z - y$.

For any expressions x and E, *x occurs in E* if either a) $x = E$ or b) $E \notin \mathbf{U}$ and x occurs in either $car(E)$ or $cdr(E)$. Clearly, if x occurs in E and $x \neq E$, then $complexity(x) < complexity(E)$. If S is a set of expressions, then the expression *x occurs in S* if x occurs in E for some $E \in S$. If $L = \langle E_n, \ldots, E_1 \rangle$ is a list of expressions and x occurs in some member E_k of L but not in any E_i such that $n \geq i > k$, then we say that *the first occurrence of x in L is in E_k*.

2 Prolog

2.1 Syntax

We shall define a language based on the four sets $\mathbf{F}$, $\mathbf{P}$, $\mathbf{K}$, and $\mathbf{V}$. The elements of $\mathbf{F}$ and $\mathbf{P}$ are called *function symbols* and *predicate symbols*, respectively. Associated with each of these is a natural number, its *arity*. While allowing predicates of arity 0, we shall assume that every function symbol has positive arity. Elements of $\mathbf{V}$ are called *variable symbols*, and elements of $\mathbf{K}$ are called *constants*. We assume that NIL is a constant and that $\mathbf{V}$ is countably infinite.

A *term* is an expression that is either a constant, a variable symbol, or a *compound term*, i.e., a pair $t = (f, a)$, where $f = funct(t)$ is a function symbol of arity n and $a = args(t)$ is a list of n terms. A *literal* is a pair $L = (p, a)$, where $p = \chi(L)$ is a predicate symbol of arity n, called the *characteristic predicate* of L, and a is a list of n terms, the *arguments* of L. For clarity, we shall sometimes write *args* in place of *cdr*.

A *Horn clause* is any list of literals. The *characteristic predicate* of a Horn clause C, $\chi(C)$, is $\chi(car(C))$, where $\chi(\texttt{NIL})$ is defined to be NIL. A Horn clause may also be called a *goal clause*, and the term *program clause* will refer to a non-`NIL` Horn clause. If $C = (h, b)$ is a program clause, then $h = head(C)$ and $b = body(C)$. The members of $body(C)$ are called the *goal literals* of C. Considered as a goal clause, a Horn clause is intended to represent the conjunction of its literals; as a program clause, it is intended to represent the disjunction of its head literal and the negations of its goal literals.

A *program* is a list of program clauses. For convenience, we shall always assume that the members of a program are distinct clauses. (Thus, the clauses of a program have a well-defined order, which will be useful in our discussion of semantics.) For any Q, the *definition* of Q with respect to a program P, denoted $def(Q, P)$, is the sublist of P consisting of all clauses of charactistic predicate Q. The list whose members are all the predicate symbols that occur in P, in the order of their first occurrence, is denoted $preds(P)$.

Let $\theta = \{(v_1, t_1), \ldots, (v_n, t_n)\}$ be a finite set of ordered pairs such that each $v_i \in \mathbf{V}$, $v_i \neq v_j$ for $i \neq j$, and each t_i is a term distinct from v_i. Then θ is called a *substitution*. Given an expression E, the expression $E\theta$ is defined as follows: if $E = v_i$ for some i, then $E\theta = t_i$; otherwise, if $E \in \mathbf{U}$, then $E\theta = E$; if $E \notin \mathbf{U}$, then $E\theta = (car(E)\theta, cdr(E)\theta)$.

For any expression E, the list of all variable symbols that occur in E, in the order of their first occurrence, is denoted $vars(E)$. A substitution θ is a *substitution for* E if for each $(v_i, t_i) \in \theta$, v_i is a member of $vars(E)$. For an arbitrary θ, the *restriction* of θ to E is the result of deleting (v_i, t_i) from θ for each v_i that does not occur in E.

If $\theta = \{(v_1, t_1), \ldots, (v_n, t_n)\}$ and $\sigma = \{(w_1, s_1), \ldots, (w_m, s_m)\}$ are two substitutions, then the *composition* $\theta\sigma$ is the substitution obtained from

$$\{(v_1, t_1\sigma), \ldots, (v_n, t_n\sigma), (w_1, s_1), \ldots, (w_m, s_m)\}$$

by deleting any pairs $(v_i, t_i\sigma)$ such that $t_i\sigma = v_i$, and any pairs (w_i, s_i) such that $w_i \in \{v_1, \ldots, v_n\}$. It is easily verified that $E(\theta\sigma) = (E\theta)\sigma$ and that if ρ is also a substitution, then $\theta(\sigma\rho) = (\theta\sigma)\rho$. Note that the empty set, $\emptyset$, is a substitution and that $E\emptyset = E$.

Let E and F be expressions. E is an *instance* of F if $E = F\theta$ for some substitution θ. E and F are *variants* if they are instances of each other. Two substitutions σ and τ for E are *equivalent for* E if $E\sigma$ and $E\tau$ are variants.

A substitution $\sigma = \{(v_1, t_1), \ldots, (v_n, t_n)\}$ is *variable pure* if each $t_i \in \mathbf{V}$. In this case, if $t_i \neq t_j$ whenever $i \neq j$, then $\hat{\sigma}$ will denote the substitution

$\{(t_1, v_1), \ldots, (t_n, v_n)\}$, and σ and $\hat{\sigma}$ are called *renamings*. If it is also the case that $\{t_1, \ldots, t_n\} - \{v_1, \ldots, v_n\}$ contains no variable symbols occurring in the expression E, then $E\sigma$ is a variant of E, since $E\sigma\hat{\sigma} = E$. In this case, if σ is a substitution for E, then σ is a *renaming for E*.

E and F are *unifiable* if $E\theta = F\theta$ for some substitution θ. θ is then called a *unifier* for E and F. θ is also called a *most general unifier (mgu)* for E and F if for any unifier σ for E and F, there is a substitution ρ such that $\sigma = \theta\rho$. It is well-known [9] that any two unifiable expressions have a mgu.

2.2 Semantics

In the sequel, P will denote a fixed program and G will denote a fixed non-`NIL` goal clause. Informally, we think of a Prolog interpreter as a procedure that accepts P and G as input and returns a sequence of substitutions called *computed answer substitutions*. The process by which these substitutions are derived is a special case of SLD-resolution [2], as described below.

Definition 2.1 *Let $H = \langle A_1, \ldots, A_m \rangle$ be a goal clause, let $C = \langle A, B_1, \ldots, B_k \rangle$ be a program clause, and let θ be a mgu for A and A_1. The goal clause $\langle B_1\theta, \ldots, B_k\theta, A_2\theta, \ldots, A_m\theta \rangle$ is called the* **resolvent** *of H and C via θ.*

Definition 2.2 *An* **SLD-derivation** *for P and G is a list $D = \langle \mathcal{G}, \mathcal{C}, \Theta \rangle$, where*

a) $\mathcal{G}$ is a list $\langle G_n, \ldots, G_0 \rangle$ of goal clauses with $G_0 = G$,

b) $\mathcal{C}$ is a list $\langle C_n, \ldots, C_1 \rangle$ of clauses of P, and

c) Θ is a list $\langle \theta_n, \ldots, \theta_1 \rangle$ of substitutions such that for $1 \leq k \leq n$, G_k is the resolvent of G_{k-1} and some variant $\tilde{C}_k$ of C_k via θ_k, and $\tilde{C}_k$ has no variables in common with either G_{k-1} or $G\theta_1 \cdots \theta_{k-1}$.

If $G_n =$ `NIL`, then D is also called an **SLD-refutation**. *In this case, the restriction to G of the composition $\theta_1 \cdots \theta_n$ is called the* **computed answer substitution** *of D. If $D' = \langle \mathcal{G}', \mathcal{C}', \Theta' \rangle$ is another SLD-derivation for P and G and $\mathcal{C} = \mathcal{C}'$, then D and D' are* **equivalent**. *The* **characteristic predicate of** *D, $\chi(D)$, is defined to be $\chi(G_n)$.*

With respect to a suitable *declarative* semantics, it may be shown [8] that an instance of G is a logical consequence of P if and only if it is an instance of $G\theta$ for some computed answer substitution θ. Our attention will be confined, however, to the *procedural* semantics of Prolog. We shall require the following technical lemma:

Lemma 2.1 *Let H_1, H_2, H_3, H_1', H_2', and H_3' be goal clauses, let C and C' be program clauses, and let α, β, θ, and θ' be substitutions such that*

a) $H_1\alpha = H_1'$, $H_2\alpha = H_2'$, and $C\beta = C'$,

b) C has no variables in common with either H_1 or H_2, and

c) H_3 (resp., H_3') is the resolvent of H_2 and C via θ (resp., H_2' and C' via θ').

Then there exists a substitution δ such that $H_3\delta = H_3'$ and $H_1\theta\delta = H_1'\theta'$.

Proof: We may assume, without loss of generality, that α is a substitution for the expression (H_1, H_2) and that β is a substitution for C. Then $\gamma = \alpha \cup \beta$ is a substitution. Moreover, $H_1\gamma = H_1'$, $H_2\gamma = H_2'$, and $C\gamma = C'$.

Since θ' is a unifier for $car(H_2')$ and $head(C')$, $\gamma\theta'$ is a unifier for $car(H_2)$ and $head(C)$. Since θ is a mgu for these expressions, we have $\gamma\theta' = \theta\delta$ for some substitution δ. Hence,

$$\begin{aligned} H_3\delta &= append(cdr(H_2)\theta, body(C)\theta)\delta \\ &= append(cdr(H_2)\theta\delta, body(C)\theta\delta) \\ &= append(cdr(H_2)\gamma\theta', body(C)\gamma\theta') \\ &= append(cdr(H_2')\theta', body(C')\theta') \\ &= H_3' \end{aligned}$$

and

$$H_1\theta\delta = H_1\gamma\theta' = H_1'\theta'.\square$$

Lemma 2.2 *Let $D = \langle\langle G_n, \ldots, G_0\rangle, \langle C_n, \ldots, C_1\rangle, \langle\theta_n, \ldots, \theta_1\rangle\rangle$ and $D' = \langle\langle G_n', \ldots, G_0'\rangle, \langle C_n, \ldots, C_1\rangle, \langle\theta_n', \ldots, \theta_1'\rangle\rangle$ be equivalent SLD-derivations for P and $G = G_0 = G_0'$.*

a) There exists a substitution δ such that $G_n\delta = G_n'$ and $G\theta_1 \cdots \theta_n\delta = G\theta_1' \cdots \theta_n'$.

b) If D is an SLD-refutation, then so is D' and the computed answer substitutions of D and D' are equivalent substitutions for G.

Proof: a) Note that for $n = 0$, the statement is satisfied by $\delta = \emptyset$. We proceed by induction, assuming that $G_{n-1}\alpha = G'_{n-1}$ and that $G\theta_1 \cdots \theta_{n-1}\alpha = G\theta'_1 \cdots \theta'_{n-1.}$.

Let $\tilde{C}_n$ and $\tilde{C}'_n$ be variants of C_n such that G_n is the resolvent of G_{n-1} and $\tilde{C}_n$ via θ_n, and G'_n is the resolvent of G'_{n-1} and $\tilde{C}'_n$ via θ'_n, in accordance with the definition of *SLD-derivation*. There exists a substitution β such that $\tilde{C}_n\beta = \tilde{C}'_n$. Thus, we have a case of Lemma 2.1, with $H_1 = G\theta_1 \cdots \theta_{n-1}$, $H_2 = G_{n-1}$, $H_3 = G_n$, $H'_1 = G\theta'_1 \cdots \theta'_{n-1.}$, $H'_2 = G'_{n-1}$, $H'_3 = G'_n$, $C = \tilde{C}_n$, and $C' = \tilde{C}'_n$.

b) By symmetry, it follows from a) that G_n and G'_n are variants, and so are $G\theta_1 \cdots \theta_n$ and $G\theta'_1 \cdots \theta'_n$.$\square$

If σ is a substitution for G, then $\bar{\sigma}$ will denote the class of substitutions for G that are equivalent to σ. The equivalence class of an SLD-derivation D will be denoted by $\bar{D}$.

Definition 2.3 *The* **SLD-tree** *for P and G is the set of all equivalence classes of SLD-derivations of P and G, ordered as follows: if $D = \langle \mathcal{G}, \mathcal{C}, \Theta \rangle$ and $D' = \langle \mathcal{G}', \mathcal{C}', \Theta' \rangle$, then $\bar{D}$ is an ancestor of $\bar{D}'$ iff $\mathcal{C}$ is a tail of $\mathcal{C}'$.*

Let $\mathcal{T}$ be the SLD-tree for P and G. The root of $\mathcal{T}$ is the class $\bar{D}_0 = \{D_0\}$, where $D_0 = \langle \langle G \rangle, \mathtt{NIL}, \mathtt{NIL} \rangle$. Any node $\bar{D}$ of $\mathcal{T}$, with $D = \langle \mathcal{G}, \mathcal{C}, \Theta \rangle$ and $\mathcal{G} = \langle G_n, \ldots, G_0 \rangle$, has a subnode corresponding to each clause of P with a variant whose head is unifiable with $car(G_n)$. This correspondence between the subnodes of $\bar{D}$ and the clauses of P, together with the order on the clauses of P, induces an order on the subnodes on $\bar{D}$, and hence a depth-first order on $\mathcal{T}$.

A Prolog interpreter performs a depth-first traversal of $\mathcal{T}$, searching for SLD-refutations and recording computed answer substitutions. More precisely,

Definition 2.4 *Let $\bar{D}_0$ be the root of the SLD-tree of P and G and let $\bar{D}_i$ be the successor of $\bar{D}_{i-1}$ with respect to the depth-first ordering for $1 \leq i \leq n$. Let $\langle \bar{D}_{i_k}, \ldots, \bar{D}_{i_0} \rangle$ be the sublist of all refutation classes of the list $\langle \bar{D}_n, \ldots, \bar{D}_0 \rangle$. Let σ_{i_j} be the computed answer substitution of D_{i_j}, for $j = 0, \ldots, k$. The list $\langle \bar{\sigma}_{i_k}, \ldots, \bar{\sigma}_{i_0} \rangle$ is called an* **interpretation** *of P and G.*

In order to compute an interpretation of P and G, we must have a means of computing the depth-first successor of a node of the SLD-tree. This is provided by the following function:

Definition 2.5 *Let $N = \bar{D}$ be a node of the SLD-tree T of P and G, where $D = \langle\langle G_n, \ldots, G_0\rangle, \langle C_n, \ldots, C_1\rangle, \langle\theta_n, \ldots, \theta_1\rangle\rangle$. Let L be a tail of $def(\chi(D), P)$. Then $next(N, L)$ is determined as follows:*

a) If $n = 0$ and $L =$ `NIL`, then $next(N, L)$ is undefined.

b) If $n > 0$ and $L =$ `NIL`, then $next(N, L) = next(N', L')$ (if defined), where $N' = parent(N)$ and $L' = cdr(member(C_n, def(\chi(N'), P)))$.

c) If $L \neq$ `NIL` and N has a subnode M corresponding to $car(L)$, then $next(N, L) = M$.

d) If $L \neq$ `NIL` and N has no subnode corresponding to $car(L)$, then $next(N, L) = next(N, cdr(L))$ (if defined).

Lemma 2.3 *Let $N = \bar{D}$ be a node of the SLD-tree $\mathcal{T}$ for P and G. Unless N is the unique depth-first maximum node of $\mathcal{T}$, the depth-first successor of N is $next(N, def(\chi(D), P))$.*

Proof: Given a sublist L of P, let $B(N, L)$ be the set of all nodes M of $\mathcal{T}$ such that either *a*) M is a descendant of a subnode of N corresponding to some member of L, or *b*) M is preceded by every descendant of N. Using induction, we shall prove that if $B(N, L) \neq \emptyset$, then $next(N, L)$ is the minimal node (with respect to the depth-first order) of $B(N, L)$. We assume that this statement is true for all N' and L', where either N' is an ancestor of N distinct from N, or $N' = N$ and L' is a tail of L distinct from L.

Case 1: $L =$ `NIL`

We may assume N is not maximal, and hence $n > 0$ and $next(N, L) = next(N', L')$, where N' and L' are as defined in case *b*) of Definition 2.5. Since every element of $B(N', L')$ is preceded by every descendant of N and therefore belongs to $B(N, L)$, it suffices to show $B(N, L) \subseteq B(N', L')$.

Let $M \in B(N, L)$. If M is a descendant of N', then since M is preceded by N, the subnode of N' of which M is a descendant corresponds to some member of L', and hence $M \in B(N', L')$. If N is not a descendant of N', then M and N' have ancestors P and Q, respectively, such that P and Q have the same parent and P precedes Q. In this case, M is preceded by all descendants of N', and again we have $M \in B(N', L')$.

Case 2: $L \neq \texttt{NIL}$

If N has a subnode M corresponding to $car(L)$, then $M = next(N, L)$ and M is the minimal node of $B(N, L)$. If N has no such subnode, then $next(N, L) = next(N, cdr(L))$ and $B(N, L) = B(N, cdr(L))$.□

2.3 Last Call Optimization

If we were to base a Prolog interpreter on a procedure that computes the function *next*, then a state of our interpreter would consist of an SLD-derivation D and a list of clauses L to be matched against the first literal of the leading goal of D. It is possible, however, to construct a somewhat more efficient interpreter, by means of a technique known as *last call optimization*. This technique is based on the observation that some of the information about the history of a computation that is contained in an SLD-derivation is irrelevant to the ultimate result. Specifically, instead of recording a clause C_i that was successfully matched with a goal G_{i-1}, we may record a list of the clauses that remain to be tried for this goal. In the event that this list is empty, the goal may be deleted from the derivation.

We conclude this section by establishing a theoretical basis for this optimization. As an alternative to SLD-derivations, we introduce the following:

Definition 2.6 *An* **interpreter state** *for P and G of length $n \geq 0$ is a list $R = \langle\langle G_n, \ldots, G_1\rangle, \langle L_n, \ldots, L_1\rangle, \langle\sigma_n, \ldots, \sigma_1\rangle\rangle$ such that for $i = 1, \ldots, n$,*

a) G_i is a goal clause, and $G_i \neq \texttt{NIL}$ if $i < n$,

b) L_i is a tail of $def(\chi(G_i), P)$, and $L_i \neq \texttt{NIL}$ if $i < n$, and

c) σ_i is a substitution for G.

R is called a **refutation state** *if $G_n = \texttt{NIL}$. In this case, σ_n is called the* **refutation substitution** *of R. If*

$$R' = \langle\langle G'_n, \ldots, G'_1\rangle, \langle L'_n, \ldots, L'_1\rangle, \langle\sigma'_n, \ldots, \sigma'_1\rangle\rangle$$

is another interpreter state for P and G, then R and R' are **equivalent** *if for $i = 1, \ldots, n$, $L_i = L'_i$ and there exist substitutions μ_i and η_i such that $G_i\mu_i = G'_i$, $G\sigma_i\mu_i = G\sigma'_i$, $G'_i\eta_i = G_i$, and $G\sigma'_i\eta_i = G\sigma_i$.*

$\bar{R}$ will denote the equivalence class of an interpreter state R for P and G. We define a transformation on the set of such equivalence classes as follows:

Definition 2.7 *Let* $R = \langle\langle G_n, \ldots, G_1\rangle, \langle L_n, \ldots, L_1\rangle, \langle\sigma_n, \ldots, \sigma_1\rangle\rangle$ *be an interpreter state for* P *and* G. *The* **successor** *of the class* $\bar{R}$ *is the class* $\bar{S} = succ(\bar{R})$, *where*

a) if $n = 1$ *and* $L_1 = \mathtt{NIL}$, *then* $\bar{S}$ *is undefined;*

b) if $n > 1$ *and* $L_n = \mathtt{NIL}$, *then* $\bar{S} = succ(\bar{R}')$, *where*

$$R' = \langle\langle G_{n-1}, \ldots, G_1\rangle, \langle L_{n-1}, \ldots, L_1\rangle, \langle\sigma_{n-1}, \ldots, \sigma_1\rangle\rangle;$$

c) if $L_n \neq \mathtt{NIL}$ *and there is no variant of* $car(L_n)$ *whose head is unifiable with* $car(G_n)$, *then* $\bar{S} = succ(\bar{R}')$, *where*

$$R' = \langle\langle G_n, \ldots, G_1\rangle, \langle cdr(L_n), L_{n-1}, \ldots, L_1\rangle, \langle\sigma_n, \ldots, \sigma_1\rangle\rangle;$$

d) if $L_n \neq \mathtt{NIL}$ *and* G_{n+1} *is the resolvent of* G_n *and* C *via* θ, *where* C *is a variant of* $car(L_n)$ *that has no variables in common with either* G_n *or* $G\sigma_n$, *then*

$$S = \langle\langle G_{n+1}, \ldots, G_1\rangle, \langle L_{n+1}, cdr(L_n), L_{n-1} \ldots, L_1\rangle, \langle\sigma_{n+1}, \ldots, \sigma_1\rangle\rangle$$

if $cdr(L_n) \neq \mathtt{NIL}$, *and*

$$S = \langle\langle G_{n+1}, G_{n-1}, \ldots, G_1\rangle, \langle L_{n+1}, L_{n-1}, \ldots, L_1\rangle, \langle\sigma_{n+1}, \sigma_{n-1}, \ldots, \sigma_1\rangle\rangle$$

if $cdr(L_n) = \mathtt{NIL}$, *where* $L_{n+1} = def(\chi(G_{n+1}), P)$ *and* σ_{n+1} *is the restriction of* $\sigma_n\theta$ *to* G.

Lemma 2.1 guarantees that this successor function is well-defined. Our goal in this section is to show that it provides an alternative model for a Prolog interpreter:

Theorem 2.1 *Let* R_0 *be the interpreter state* $\langle\langle G\rangle, \langle L\rangle, \langle\emptyset\rangle\rangle$ *for* P *and* G, *where* $L = def(\chi(G), P)$. *For* $1 \leq i \leq n$, *let* $\bar{R}_i$ *be the successor of* $\bar{R}_{i-1}$. *Let* $\langle\bar{R}_{i_k}, \ldots, \bar{R}_{i_0}\rangle$ *be the sublist of the list* $\langle\bar{R}_n, \ldots, \bar{R}_0\rangle$ *consisting of all classes of refutation states. Let* σ_{i_j} *be the refutation substitution of* R_{i_j}, *for* $j = 0, \ldots, k$. *Then the list* $\langle\bar{\sigma}_{i_k}, \ldots, \bar{\sigma}_{i_0}\rangle$ *is an interpretation of* P *and* G.

The proof of Theorem 2.1 is based on a mapping from SLD-derivations to interpreter states:

Definition 2.8 *Let $D = \langle\langle G_n, \ldots, G_0\rangle, \langle C_n, \ldots, C_1\rangle, \langle\theta_n, \ldots, \theta_1\rangle\rangle$ be an SLD-derivation of P and G, and let L be a tail of $def(\chi(D), P)$. The* **reduction** *of D and L is the interpreter state $R = red(D, L)$, defined as follows:*

a) If $n = 0$, then $R = \langle \mathtt{NIL}, \mathtt{NIL}, \mathtt{NIL}\rangle$ if $L = \mathtt{NIL}$, and $R = \langle\langle G\rangle, \langle L\rangle, \langle\emptyset\rangle\rangle$ otherwise.

b) If $n > 0$, then let $R' = \langle\langle H_k, \ldots, H_1\rangle, \langle L_k, \ldots, L_1\rangle, \langle\sigma_k, \ldots, \sigma_1\rangle\rangle$ be the reduction of $\langle\langle G_{n-1}, \ldots, G_0\rangle, \langle C_{n-1}, \ldots, C_1\rangle, \langle\theta_{n-1}, \ldots, \theta_1\rangle\rangle$ and L', where $L' = cdr(member(C_n, def(\chi(G_{n-1}), P)))$. If $L = \mathtt{NIL}$ and $def(\chi(D), P) \neq \mathtt{NIL}$, then $R = R'$; otherwise,

$$R = \langle\langle G_n, H_k, \ldots, H_1\rangle, \langle L, L_k, \ldots, L_1\rangle, \langle\sigma, \sigma_k, \ldots, \sigma_1\rangle\rangle,$$

where σ is the restriction of $\theta_1 \cdots \theta_n$ to G.
In the case $L = def(\chi(D), P)$, $red(D, L)$ will also be denoted by $opt(D)$.

As an immediate result of this definition, we have

Lemma 2.4 *Except in the case where $L = \mathtt{NIL}$ and $def(\chi(D), P) \neq \mathtt{NIL}$, $red(D, L)$ and D have the same leading goal clause. $red(D, L)$ is a refutation state iff D is an SLD-refutation. If D is an SLD-refutation, then the computed answer substitution of D is the refutation substitution of $red(D, L)$.*

The following is a consequence of Lemma 2.2 and Definitions 2.6 and 2.8:

Lemma 2.5 *Let D and D' be equivalent SLD-derivations for P and G. Let L be a tail of $def(\chi(D), P)$. Then $red(D, L)$ and $red(D', L)$ are equivalent interpreter states.*

Lemma 2.5 implies that the functions *red* and *opt* induce functions $\overline{red}$ and $\overline{opt}$, defined on the SLD-tree of P and G:

Definition 2.9 *Let $\bar{D}$ be a node of the SLD-tree for P and G and let L be a tail of $def(\chi(D), P)$.*

a) If $R = red(D, L)$, then $\overline{red}(\bar{D}, L) = \bar{R}$.

b) If $R = opt(D)$, then $\overline{opt}(\bar{D}) = \bar{R}$.

Theorem 2.1 will be derived from

Lemma 2.6 *Let* $D = \langle\langle G_n, \ldots, G_0\rangle, \langle C_n, \ldots, C_1\rangle, \langle\theta_n, \ldots, \theta_1\rangle\rangle$ *be an SLD-derivation for* P *and* G *and let* L *be a tail of* $def(\chi(D), P)$*. If* $next(\bar{D}, L)$ *is defined, then*

$$\overline{opt}(next(\bar{D}, L)) = succ(\overline{red}(\bar{D}, L)).$$

Proof: We follow the same induction scheme as in Lemma 2.3.

Case 1: $L = \mathtt{NIL}$

In this case, $next(\bar{D}, L) = next(\bar{D}', L')$, where

$$D' = \langle\langle G_{n-1}, \ldots, G_0\rangle, \langle C_{n-1}, \ldots, C_1\rangle, \langle\theta_{n-1}, \ldots, \theta_1\rangle\rangle$$

and $L' = cdr(member(C_n, def(\chi(D'), P)))$. Let

$$red(D', L') = \langle\langle H_k, \ldots, H_1\rangle, \langle L_k, \ldots, L_1\rangle, \langle\sigma_k, \ldots, \sigma_1\rangle\rangle.$$

If $def(\chi(D), P) = \mathtt{NIL}$, then

$$red(D, L) = \langle\langle G_n, H_k, \ldots, H_1\rangle, \langle \mathtt{NIL}, L_k, \ldots, L_1\rangle, \langle\sigma, \sigma_k, \ldots, \sigma_1\rangle\rangle$$

and $succ(\overline{red}(\bar{D}, L)) = succ(\overline{red}(\bar{D}', L'))$. If $def(\chi(D), P) \neq \mathtt{NIL}$, then $red(D, L) = red(D', L')$. In either case,

$$\begin{aligned} \overline{opt}(next(\bar{D}, L)) &= \overline{opt}(next(\bar{D}', L')) \\ &= succ(\overline{red}(\bar{D}', L')) \\ &= succ(\overline{red}(\bar{D}, L)). \end{aligned}$$

Case 2: $\bar{D}$ *has a subnode corresponding to* $car(L)$*.*

Since $L \neq \mathtt{NIL}$, we may again write

$$red(D, L) = \langle\langle G_n, H_k, \ldots, H_1\rangle, \langle L, L_k, \ldots, L_1\rangle, \langle\sigma, \sigma_k, \ldots, \sigma_1\rangle\rangle$$

for some $k \geq 0$, where σ is the restriction of $\theta_1 \cdots \theta_n$ to G. $next(\bar{D}, L) = \bar{D}'$, for some

$$D' = \langle\langle G_{n+1}, \ldots, G_0\rangle, \langle C_{n+1}, \ldots, C_1\rangle, \langle\theta_{n+1}, \ldots, \theta_1\rangle\rangle,$$

where $C_{n+1} = car(L)$ and G_{n+1} is the resolvent of G_n and some variant of C_{n+1} via θ_{n+1}.

Consider the subcase $cdr(L) \neq \mathtt{NIL}$. If $L' = def(\chi(D'), P)$, then $red(D', L')$ is

$$\langle\langle G_{n+1}, G_n, H_k, \ldots, H_1\rangle, \langle L', cdr(L), L_k, \ldots, L_1\rangle, \langle \rho, \sigma, \sigma_k, \ldots, \sigma_1\rangle\rangle,$$

where ρ is the restriction of $\theta_1 \cdots \theta_{n+1}$ to G. The successor of $\overline{red}(\bar{D}, L)$ is the class of

$$\langle\langle G_{n+1}, G_n, H_k, \ldots, H_1\rangle, \langle L', cdr(L), L_k, \ldots, L_1\rangle, \langle \tau, \sigma, \sigma_k, \ldots, \sigma_1\rangle\rangle,$$

where τ is the restriction of $\sigma\theta_{n+1}$ to G. Since $G\tau = G\sigma\theta_{n+1} = G\rho$,

$$\begin{aligned} \overline{opt}(next(\bar{D}, L)) &= \overline{red}(next(\bar{D}, L), L') \\ &= \overline{red}(\bar{D}', L') \\ &= succ(\overline{red}(\bar{D}, L)). \end{aligned}$$

The subcase $cdr(L) = \mathtt{NIL}$ is the same, except that $\overline{red}(\bar{D}', L') = succ(\overline{red}(\bar{D}, L))$ is the class of

$$\langle\langle G_{n+1}, H_k, \ldots, H_1\rangle, \langle L', L_k, \ldots, L_1\rangle, \langle \tau, \sigma_k, \ldots, \sigma_1\rangle\rangle.$$

Case 3: *$L \neq \mathtt{NIL}$ and $\bar{D}$ has no subnode corresponding to $car(L)$.*

In this case, $next(\bar{D}, L) = next(\bar{D}, cdr(L))$ and

$$\overline{opt}(next(\bar{D}, L)) = \overline{opt}(next(\bar{D}, cdr(L))).$$

Let

$$red(D, L) = \langle\langle G_n, H_k, \ldots, H_1\rangle, \langle L, L_k, \ldots, L_1\rangle, \langle \sigma, \sigma_k, \ldots, \sigma_1\rangle\rangle.$$

If $cdr(L) \neq \mathtt{NIL}$, then

$$red(D, cdr(L)) = \langle\langle G_n, H_k, \ldots, H_1\rangle, \langle cdr(L), L_k, \ldots, L_1\rangle, \langle \sigma, \sigma_k, \ldots, \sigma_1\rangle\rangle,$$

otherwise

$$red(D, cdr(L)) = \langle\langle H_k, \ldots, H_1\rangle, \langle L_k, \ldots, L_1\rangle, \langle \sigma_k, \ldots, \sigma_1\rangle\rangle.$$

In either case, $succ(\overline{red}(\bar{D}, L)) = succ(\overline{red}(\bar{D}, cdr(L)))$, and the result follows from the inductive hypothesis.$\square$

Proof of Theorem 2.1: If R_0 is the interpreter state $\langle\langle G\rangle, \langle L_0\rangle, \langle\emptyset\rangle\rangle$, where $L_0 = def(\chi(G), P)$, then $R_0 = opt(D_0)$, where $D_0 = \langle\langle G\rangle, \mathtt{NIL}, \mathtt{NIL}\rangle$. For

$i = 1, \ldots, n$, let $L_i = def(\chi(D_{i-1}), P)$ and $D_i = next(D_{i-1}, L_{i-1})$. By induction on n, we have

$$\begin{aligned} \bar{R}_i &= succ(\bar{R}_{i-1}) \\ &= succ(\bar{red}(\bar{D}_{i-1}, L_{i-1})) \\ &= \overline{opt}(next(\bar{D}_{i-1}, L_{i-1})) \\ &= \overline{opt}(\bar{D}_i). \end{aligned}$$

The theorem now follows from Lemma 2.4.□

3 The WAM

3.1 WAM States

We assume that $\mathbf{R} = \{\mathbf{h}, \mathbf{b}, \mathbf{e}, \mathbf{t}, \mathbf{a}, \mathbf{p}, \mathbf{c}, \mathbf{s}\}$ is a set of order 8, and each of the sets $\mathbf{A}$, $\mathbf{S}$, $\mathbf{H}$, $\mathbf{T}$, and $\mathbf{C}$ is countably infinite: $\mathbf{A} = \{A_1, A_2, \ldots\}$, $\mathbf{S} = \{S_1, S_2, \ldots\}$, $\mathbf{H} = \{H_1, H_2, \ldots\}$, $\mathbf{T} = \{T_1, T_2, \ldots\}$, and $\mathbf{C} = \{C_1, C_2, \ldots\}$. Our formal construction of the Warren machine architecture is based on the set

$$\mathbf{M} = \mathbf{R} \cup \mathbf{A} \cup \mathbf{S} \cup \mathbf{H} \cup \mathbf{T} \cup \mathbf{C},$$

called the *WAM memory.*

An element of $\mathbf{M}$ is called a *cell.* Each element of $\mathbf{R}$ is called a *register* and each element of $\mathbf{A}$ is an *argument register*, also called a *temporary variable.* The sets $\mathbf{S}$, $\mathbf{H}$, $\mathbf{T}$, and $\mathbf{C}$ are the *local stack*, *heap*, *trail*, and *code area*, respectively. We shall refer to the subscript of a cell in $\mathbf{M} - \mathbf{R}$ as its *address.* (For example, H_k is the cell at address k in the heap.)

Machine states will be formalized as functions defined on the domain $\mathbf{M}$. The values of these functions will represent the contents of memory cells:

Definition 3.1 *A* **WAM state** *is a function $W : \mathbf{M} \to \mathbf{U}$ such that $W(r) \in \mathbf{N}$ for each $r \in \mathbf{R}$. If W is a WAM state and $m \in \mathbf{M}$, then $W(m)$ is called the* **contents** *of m with respect to W.*

The execution of a WAM program will be defined as a sequence of WAM states, each of which is derived from the preceding state by applying a state transformation that corresponds to some machine instruction. Each of these transformations will be defined in terms of the following primitive operation, by which a value is *stored* in a memory cell:

Definition 3.2 *Let* $c \in \mathbf{M}$ *and let* W *be a WAM state. For any* $x \in \mathbf{U}$*, the state* $W[c \leftarrow x]$ *is defined by*

$$W[c \leftarrow x](m) = \begin{cases} x & \text{if } m = c \\ W(m) & \text{if } m \in \mathbf{M} - \{c\} \end{cases}$$

$W[c_1 \leftarrow x_1] \cdots [c_r \leftarrow x_r]$ *will also be written as* $W[c_1 \leftarrow x_1, \ldots, c_r \leftarrow x_r]$.

As described below, each register in $\mathbf{R}$ is associated with one of the other WAM memory areas: $\mathbf{H}$, $\mathbf{A}$, $\mathbf{S}$, $\mathbf{T}$, or $\mathbf{C}$. The contents of a register with respect to a WAM state is a natural number, to be interpreted as an address in its associated area.

The registers $\mathbf{p}$ and $\mathbf{c}$ are both associated with the code area $\mathbf{C}$, where *WAM instructions* (Section 3.7) are stored. The register $\mathbf{p}$, called the *program counter*, generally contains the address in $\mathbf{C}$ that corresponds to the next instruction to be executed. The value 0, however, has special significance:

Definition 3.3 *A WAM state* W *is a* **success state** *if* $W(\mathbf{p}) = 0$.

We define the following operations pertaining to $\mathbf{p}$:

Definition 3.4 *Let* W *be a WAM state.*

a) $jump(p, W) = W[\mathbf{p} \leftarrow p]$

b) $advance(n, W) = W[\mathbf{p} \leftarrow W(\mathbf{p}) + n]$

c) $read(n, W) = W(C_{W(\mathbf{p})+n})$

The set of clauses comprising the definition of a Prolog predicate is compiled into a list called a *WAM procedure* (Section 4.4). Each of these procedures corresponds to an address in the code area. One procedure may pass control to another by storing its address in the program counter by means of the *jump* operation. (These notions of *procedure* and *control* shall remain informal and will be used only to motivate our definitions.)

The *continuation pointer*, $\mathbf{c}$, is used to store return addresses for procedure calls. Thus, immediately before jumping to the called procedure, the current value of the program counter is stored in $\mathbf{c}$:

Definition 3.5 *For any WAM state W,*

$$\textit{set-return-address}(W) = W[\mathbf{c} \leftarrow W(\mathbf{p})].$$

Upon successful termination of the called procedure, the return address is copied back into the program counter:

Definition 3.6 *If W is a WAM state, then*

$$\textit{proceed}(W) = \textit{jump}(W(\mathbf{c}), W).$$

The areas **H**, **T**, and **S**, which are used to store *WAM objects* (Section 3.2), are all manipulated as stacks. The stack pointers associated with **H** and **T** are the *heap pointer* **h** and the *trail pointer* **t**, respectively. The local stack **S** has two associated stack pointers: the *environment pointer* **e** and the *backtrack pointer* **b**. These registers correspond to the two distinct types of structures that are stored in **S**: *environments* (Section 3.3) and *choice points* (Section 3.4), respectively. The top of **S** is determined by the maximum of the contents of these two pointers.

The *structure pointer*, **s**, is a second register associated with the heap. It contains a heap address that is incremented by the WAM unification instructions (Section 3.6) by means of the following:

Definition 3.7 *If W is a WAM state, then*

$$\textit{next-arg}(W) = W[\mathbf{s} \leftarrow W(\mathbf{s}) + 1].$$

The *arity register* **a** is associated with the area **A**, which is used to store arguments to WAM procedures. When a procedure is called, its arguments are passed in some initial segment $\{A_1, \ldots, A_n\}$ of **A**. The number n, which is the arity of the Prolog predicate that produced this procedure, is stored in **a** when the procedure is called:

Definition 3.8 *If W is a WAM state and q is a predicate symbol of arity n, then*

$$\textit{set-arity}(q, W) = W[\mathbf{a} \leftarrow n].$$

With respect to any given WAM state, only a finite number of cells are of interest, determined in part by the contents of various registers:

Definition 3.9 *Two WAM states W and W' are* **equivalent***, and we write $W \approx W'$, if $W(m) = W'(m)$ for each $m \in \mathbf{M}$ that satisfies any of the following:*

a) $m \in \{\mathbf{h}, \mathbf{t}, \mathbf{a}, \mathbf{b}, \mathbf{e}, \mathbf{c}, \mathbf{p}\}$	*d)* $m = A_k,\ k \leq W(\mathbf{a})$
b) $m = H_k,\ k \leq W(\mathbf{h})$	*e)* $m = S_k,\ k \leq max(W(\mathbf{b}), W(\mathbf{e}))$
c) $m = T_k,\ k \leq W(\mathbf{t})$	*f)* $m \in \mathbf{C}$

3.2 WAM Objects

In the context of a WAM state, Prolog terms are represented as memory cells in the stack and in the heap. An *unbound variable*, which is the internal representation of a variable symbol, is a cell that contains a reference to itself, while a *bound variable* contains a reference to its *binding*. Compound terms are represented by cells in the heap called *structures*. More precisely,

Definition 3.10 *Let W be a WAM state. ξ is an* **object** *of W if any of the following holds:*

a) ξ is a constant.

b) $\xi = H_k$, $k \leq W(\mathbf{h})$, and either $W(\xi) = \xi$ or $W(\xi)$ is an object of W. In this case, ξ is a **heap variable** *of W.*

c) $\xi = S_k$, $k \leq W(\mathbf{e})$, and either $W(\xi) = \xi$ or $W(\xi)$ is an object of W. In this case, ξ is a **permanent variable** *of W.*

d) $\xi = H_k$, $k \leq W(\mathbf{h})$, $W(\xi) = f$ is a function symbol of arity n, and H_{k+j} is a heap variable of W for $j = 1, \ldots, n$. In this case, ξ is a **structure** *of W with* **functor** *f and* **arguments** *$H_{k+1}, \ldots, H_{k+n}$. The list $\langle H_{k+1}, \ldots, H_{k+n} \rangle$ is denoted* $\mathbf{args}(\xi, W)$.

A heap variable or permanent variable of W is **unbound** *if $W(\xi) = \xi$. Otherwise, ξ is* **bound** *and $W(\xi)$ is its* **binding***.*

Thus, each WAM object is constructed from constants and unbound variables by a finite number of instantiations of Definition 3.10. A number of inductive arguments concerning these objects will be based on the following:

Definition 3.11 *For any object ξ of a WAM state W, $count(\xi, W)$ is*

a) 1, if ξ is a constant or an unbound variable of W,

b) $1 + count(\beta, W)$, if ξ is a variable of W with binding β, *c)* $1 + \sum_{i=1}^{n} count(a_i, W)$*, if* ξ *is a structure of* W *and* $args(\xi, W) = \langle a_1, \ldots, a_n \rangle$.

Definition 3.12 *Let* v *be a variable of a state* W*, and let* ξ *be an object of* W*. Then* v **occurs in** ξ *with respect to* W *if one of the following conditions holds:*

a) $\xi = v$.

b) ξ *is a bound variable of* W *and* v *occurs in the binding of* ξ.

c) ξ *is a structure of* W *and* v *occurs in one of the arguments of* ξ.

As an immediate consequence of these definitions, we have

Lemma 3.1 *Let* v *be a variable of a state* W*, and let* $\xi \neq v$ *be an object of* W*. If* v *occurs in* ξ *with respect to* W*, then* $count(v, W) < count(\xi, W)$.

The process of *variable dereferencing* is a basic WAM operation, essential to the unification procedure described in Section 3.6:

Definition 3.13 *Let* x *be an object of a state* W*. The* **dereferenced value** *of* x *with respect to* W *is*

$$deref(x, W) = \begin{cases} deref(W(x), W) & \textit{if } x \textit{ is a bound variable of } W \\ x & \textit{otherwise} \end{cases}$$

In order to interpret the objects of a state W as Prolog terms, we must first assign a variable symbol to each unbound variable of W:

Definition 3.14 *Let* N *be an injection from the set of variables of a WAM state* W *into the set* $\mathbf{V}$ *of variable symbols and let* x *be an object of* W*. The* **term represented by** x *with respect to* W *and* N*, denoted by* $[x]_{W,N}$*, is*

a) x*, if* x *is a constant*

b) $N(x)$*, if* x *is an unbound variable*

c) $[deref(x, W)]_{W,N}$*, if* x *is a bound variable*

d) $\langle f, [a_1]_{W,N}, \ldots, [a_n]_{W,N} \rangle$*, if* x *is a structure with functor* f *and arguments* $a_1, \ldots, a_n$.

For convenience, we shall let $N : \mathbf{H} \cup \mathbf{S} \rightarrow \mathbf{V}$ be an arbitrary fixed injection such that the range of N contains no variable symbols occurring in either P or G. If x is an object of a state W, then $[x]_{W,N}$ will be abbreviated as $[x]_W$.

By a simple inductive argument based on the *count* function, we have

Lemma 3.2 *Let v be an unbound variable of a state W, and let ξ be an object of W. Then v occurs in ξ with respect to W iff $[v]_W = N(v)$ occurs in $[\xi]_W$.*

3.3 Environments

One of the two uses of the local stack is to store structures called *environments*. An environment contains data used by a WAM procedure that must be preserved across a call to another procedure.

Since the contents of the continuation pointer $\mathbf{c}$ are generally altered by a procedure call, the contents of $\mathbf{c}$ before the call (i.e., the return address of the calling procedure) is one of the data that must be saved in an environment. Since the allocation of the environment itself alters the contents of the environment pointer $\mathbf{e}$ (i.e., the address of the most recently created environment), the prior contents of $\mathbf{e}$ must also be saved. Along with these two data, an environment may contain an arbitrary number of variables, which are created when the environment is allocated:

Definition 3.15 *Let W be a WAM state, let $k = max(W(\mathbf{b}), W(\mathbf{e}))$, and let $n \in \mathbf{N}$. Then allocate(n, W) is the state*

$$\begin{aligned} W[&S_{k+1} \leftarrow S_{k+1}, \ldots, S_{k+n} \leftarrow S_{k+n}, S_{k+n+1} \leftarrow W(\mathbf{c}), \\ &S_{k+n+2} \leftarrow W(\mathbf{e}), \mathbf{e} \leftarrow k+n+2]. \end{aligned}$$

This operation creates n new unbound variables on the local stack, then pushes the contents of $\mathbf{c}$ and the old contents of $\mathbf{e}$, leaving $\mathbf{e}$ pointing to the new stack top. The resulting state is characterized by

Definition 3.16 *Let W be a WAM state with $k = W(\mathbf{e})$ and let $n \in \mathbf{N}$. W* **has an environment of order** *n if $W(S_k) \in \mathbf{N}$, $W(S_{k-1}) \in \mathbf{N}$, $W(S_k) \leq k-2-n$, and for $i = 1, \ldots, n$, S_{k-1-i} is a variable of W, denoted by $pv(i, W)$.*

The inverse of the *allocate* operation is given by

Definition 3.17 *If W is a WAM state that has an environment (of any order) and $k = W(\mathbf{e})$, then*

$$deallocate(W) = W[\mathbf{e} \leftarrow W(S_k), \mathbf{c} \leftarrow W(S_{k-1})].$$

Lemma 3.3 *Let W be a WAM state and let $W' = allocate(n, W)$ for some $n \in \mathbf{N}$. Then W' has an environment of order n and $deallocate(W') \approx W$.*

For a state W that has an environment, we would like to distinguish between the objects of W that are also objects of $deallocate(W)$ and those that are not. Thus,

Definition 3.18 *Let W be a WAM state with an environment, let $k = W(\mathbf{e})$, and let ξ be an object of W. If $\xi = S_j$, $W(S_k) < j < k$, then ξ is an* **unstable** *object of W; otherwise, ξ is a* **stable** *object of W.*

In particular, if W has an environment of order n, then each of the permanent variables $pv(i, W)$, $i = 1, \ldots, n$, is unstable. It is clear that any object of $deallocate(W)$ must be a stable object of W. However, if an unstable variable of W occurs in a stable object of W, then the converse is false. In Section 3.5, we shall introduce a constraint on WAM states that will preclude this situation.

3.4 Choice Points

The second function of the local stack is to store structures called *choice points*. A choice point is a block of information that essentially characterizes a WAM state. As the execution of a WAM program proceeds from this state, the choice point remains intact, so that it may be used later to reconstruct the state from which it was derived. The process of reconstruction is called *backtracking*.

A choice point is constructed by pushing the contents of various registers and argument registers onto the local stack, along with a code area address. The address of the new stack top is then saved as the contents of the backtrack pointer. All of this is performed by the following transformation:

Definition 3.19 *Let* W *be a WAM state,* $a = W(\mathbf{a})$*,* $k = max(W(\mathbf{b}), W(\mathbf{e}))$*, and* $p \in \mathbf{N}$*. Then try-me-else(p, W) is the state*

$$W[S_{k+1} \leftarrow W(A_1), \ldots, S_{k+a} \leftarrow W(A_a), S_{k+a+1} \leftarrow a, S_{k+a+2} \leftarrow W(\mathbf{e}),\\ S_{k+a+3} \leftarrow W(\mathbf{t}), S_{k+a+4} \leftarrow W(\mathbf{c}), S_{k+a+5} \leftarrow W(\mathbf{h}), S_{k+a+6} \leftarrow W(\mathbf{b}),\\ S_{k+a+7} \leftarrow p, \mathbf{b} \leftarrow k + a + 7].$$

The goal of backtracking is to restore the state that existed immediately after the last choice point was created. One aspect of this process is the unbinding of variables that have become bound since the creation of the choice point. The purpose of the trail $\mathbf{T}$ is to record these variables as they become bound. The first phase of backtracking is the *unwinding* of the trail, i.e., the unbinding of variables that are read from the top of the trail $W(\mathbf{t})$ down to the trail address t that is stored in the choice point:

Definition 3.20 *Let* W *be a WAM state and let* $0 \leq t \leq W(\mathbf{t})$*. Suppose that* $\xi_i = W(T_i)$ *is a variable of* W *for* $i = t+1, \ldots, W(\mathbf{t})$*. Then*

$$unwind(t, W) = W[\mathbf{t} \leftarrow t, \xi_{W(\mathbf{t})} \leftarrow \xi_{W(\mathbf{t})}, \ldots, \xi_{t+1} \leftarrow \xi_{t+1}].$$

Along with the unwinding of the trail, backtracking consists of copying values from the local stack into various registers and argument registers:

Definition 3.21 *Let* W *be a WAM state and let* $k = W(\mathbf{b})$*. Suppose that* $k \geq 7$ *and that* $p = W(S_k)$*,* $b = W(S_{k-1})$*,* $h = W(S_{k-2})$*,* $c = W(S_{k-3})$*,* $t = W(S_{k-4})$*,* $e = W(S_{k-5})$*, and* $a = W(S_{k-6})$ *are all natural numbers, with* $b < k$*,* $h \leq W(\mathbf{h})$*,* $t \leq W(\mathbf{t})$*,* $e < k$*, and* $a \leq k-7$*. Let* $x_i = W(S_{k-7-a+i})$ *for* $i = 1, \ldots, a$*. Suppose also that* $W(T_i)$ *is a variable of* W *for* $t < i \leq W(\mathbf{t})$*. Then* W **has a choice point** *and backtrack(W) is the state*

$$unwind(t, W)[\mathbf{p} \leftarrow p, \mathbf{h} \leftarrow h, \mathbf{c} \leftarrow c, \mathbf{e} \leftarrow e, \mathbf{a} \leftarrow a, A_1 \leftarrow x_1, \ldots, A_a \leftarrow x_a].$$

An obvious consequence of this definition is

Lemma 3.4 *If* W *is a WAM state and* $W' = $ *try-me-else(p, W), then* W' *has a choice point and* $backtrack(W') = jump(p, W')$*.*

The main property of the backtracking operation is that it is unaffected by composition with any of various other state transformations:

Lemma 3.5 *Let W be a WAM state that has a choice point and let $n \in \mathbf{N}$. In each of the following cases, the state W' also has a choice point and $backtrack(W') \approx backtrack(W)$:*

a) $W' \approx W$

b) $W' = W[m \leftarrow n]$, *where* $m \in \{\mathbf{a}, \mathbf{c}, \mathbf{p}\}$

c) $W' = W[\mathbf{h} \leftarrow n]$ *and* $n \geq W(\mathbf{h})$

e) $W' = allocate(n, W)$

f) *W has an environment and* $W' = deallocate(W)$.

We shall require two other operations pertaining to choice points: one for altering the code address stored in a choice point, and another for removing a choice point from the local stack. Since $S_{W(\mathbf{b})}$ is the cell that contains the code address component of the choice point of a state W, the first of these operations is given by

Definition 3.22 *If W is a WAM state that has a choice point, then*

$$retry\text{-}me\text{-}else(p, W) = W[S_{W(\mathbf{b})} \leftarrow p]$$

Since $S_{W(\mathbf{b})-1}$ contains the contents of $\mathbf{b}$ prior to the creation of the choice point, the following operation effectively removes a choice point:

Definition 3.23 *If W is a WAM state that has a choice point, then*

$$trust\text{-}me\text{-}else\text{-}fail(W) = W[\mathbf{b} \leftarrow W(S_{W(\mathbf{b})-1})]$$

As immediate consequences of the above definitions, we have

Lemma 3.6 *If W has a choice point, then*

$$backtrack(retry\text{-}me\text{-}else(p, W)) = jump(p, backtrack(W)).$$

Lemma 3.7 *If $W' \approx try\text{-}me\text{-}else(p, W)$, then $trust\text{-}me\text{-}else\text{-}fail(W') \approx W$.*

3.5 Binding Variables

We would like to define the operation of variable-binding in such a way that it is reversed by backtracking. In order to allow this, variables are recorded on the trail as they are bound. Note that an unbound variable v of W (either in the local stack or in the heap) must be trailed only if W has a choice point and v is also a variable of $backtrack(W)$. Since the local stack top and the heap top of $backtrack(W)$ are stored in the local stack at the addresses $W(\mathbf{b})$ and $W(S_{W(\mathbf{b})-2})$, respectively, our binding operation is defined as follows:

Definition 3.24 *Let x be an object of a WAM state W and let v be an argument register or an unbound variable of W. If W has a choice point and either $v = H_k$, $k \leq W(S_{W(\mathbf{b})-2})$, or $v = S_k$, $k \leq W(\mathbf{b})$, then*

$$bind(v, x, W) = W[\mathbf{t} \leftarrow W(\mathbf{t}) + 1, T_{W(\mathbf{t})+1} \leftarrow v, v \leftarrow x].$$

In all other cases,

$$bind(v, x, W) = W[v \leftarrow x].$$

The above definition is designed so that any variable that is bound after the creation of the choice point either becomes nonexistent or is unbound by backtracking. Thus, we have

Lemma 3.8 *Let W be a WAM state and let v be an argument register or an unbound variable of W. If W has a choice point, then*

$$backtrack(bind(v, x, W)) \approx backtrack(W).$$

Recall that the count of a variable exceeds that of its binding and must be exceeded by the count of any object in which it occurs. Thus, if an unbound variable v of W occurs in an object x, then v cannot be a variable of the state $bind(v, x, W)$. However, by induction on $count(x, W)$, we may easily prove

Lemma 3.9 *Let x be an object of a WAM state W and let v be an argument register or an unbound variable of W. If v does not occur in x, then every object of W is an object of $bind(v, x, W)$.*

When two unbound variables are unified (Section 3.6), a choice is made as to which variable is bound to the other. The primary concern is to avoid any state in which a stable variable is bound to an unstable one, since deallocation in this situation would result in a dangling reference. This is ensured by the requirement that no heap variable is bound to a stack variable, and that no stack variable is bound to a stack variable of higher address.

A secondary concern is one of efficiency: due to the expense of trailing and untrailing, it is generally desirable to bind the variable that is less likely to require trailing, i.e., to survive backtracking. Thus, when two heap variables are unified, we bind the one with higher address.

The following definition provides a heuristic that governs the choice of binding direction:

Definition 3.25 *If u and v are elements of $\mathbf{H} \cup \mathbf{S}$, then u* **precedes** *v if any of the following holds:*

a) $u \in \mathbf{H}$ and $v \in \mathbf{S}$

b) $u = H_k$, $v = H_l$, and $k < l$

c) $u = S_k$, $v = S_l$, and $k < l$

During the execution of a Prolog program, no variable will be bound to a variable that does not precede it. Thus, the following property of WAM states will be preserved:

Definition 3.26 *A WAM state W is* **directed** *if for each variable v of W, if $W(v)$ is a variable of W, then $W(v)$ precedes v.*

As a result, no dangling references will be created by deallocation. More precisely,

Lemma 3.10 *Let W be a directed WAM state with an environment and let ξ be an object of W. ξ is a stable object of W iff ξ is an object of $deallocate(W)$.*

It is sometimes desirable to construct a state in which two variables (perhaps temporary variables) have the same binding, without binding either variable to the other. This is achieved by binding both variables to a new variable in the heap:

Definition 3.27 *Let W be a WAM state and let each of u and v be either an argument register or an unbound variable of W. Let $h = W(\mathbf{h}) + 1$ and let $W' = W[H_h \leftarrow H_h, \mathbf{h} \leftarrow h]$. Then*

$$stabilize(u, v, W) = bind(v, H_h, bind(u, H_h, W')).$$

We note two obvious properties of this transformation:

Lemma 3.11 *Let W be a WAM state and let each of u and v be either an argument register or an unbound permanent variable of W. Let $W' = stabilize(u, v, W)$.*

a) $backtrack(W') \approx backtrack(W)$.

b) If W is directed, then so is W'.

3.6 Unification

A number of WAM instructions are based on the *unify* transformation, as defined in Figure 1. The arguments of $unify$ are a WAM state W and two objects x and y of W. If the Prolog terms represented by x and y are unifiable, then $unify(x, y, W)$ is a state in which x and y represent the same term, produced by performing a series of variable bindings. On the other hand, if these terms are not unifiable, then the state $unify(x, y, W)$ is equivalent to $backtrack(W)$.

The definition of *unify* is based on the auxiliary function *unify-lists*, the arguments of which are a WAM state W and two lists of objects l and m of W. In order to see that the recursive definition of *unify-lists* is valid, i.e., that it is satisfied by a unique function, observe that on each recursive call, either W is replaced by a state with fewer unbound variables than W, or W is held constant and $l = \langle u_n, \ldots, u_1 \rangle$ is replaced by a list $l' = \langle u'_{n'}, \ldots, u'_1 \rangle$ such that

$$\sum_{i=1}^{n'} count(u'_i, W) < \sum_{i=1}^{n} count(u_i, W).$$

The following definition will be useful in our characterization of $unify$:

Definition 3.28 *Let W and U be directed WAM states and let θ be a substitution. U is an* **extension** *of W by θ if*

a) $U(\mathbf{c}) = W(\mathbf{c})$, $U(\mathbf{e}) = W(\mathbf{e})$, and $U(C_k) = W(C_k)$ for all $k \geq 1$

Function *unify-lists(l,m,W)*;
$\{l$ and m are lists of equal length of objects of a WAM state $W\}$
if $l =$ `NIL` **then return** W
else begin
  $x \leftarrow deref(car(l), W)$; $y \leftarrow deref(car(m), W)$;
  if $x = y$ **then return** *unify-lists(cdr(l),cdr(m),W)*
  else if x is a variable **then**
    if y is a variable **then**
      if x precedes y
        then return *unify-lists(cdr(l),cdr(m),bind(y,x,W))*
        else return *unify-lists(cdr(l),cdr(m),bind(x,y,W))*
    else if x occurs in y w.r.t. W **then return** $backtrack(W)$
    else return *unify-lists(cdr(l),cdr(m),bind(x,y,W))*
  else if y is a variable **then**
    if y occurs in x w.r.t. W
      then return $backtrack(W)$
      else return *unify-lists(cdr(l),cdr(m),bind(y,x,W))*
  else if x and y are structures with the same functor **then**
    return *unify-lists(append(args(x,W),cdr(l)),append(args(y,W),cdr(m)),W)*
  else return $backtrack(W)$
end.

Function $unify(x, y, W)$;
$\{x$ and y are objects of a WAM state $W\}$
return *unify-lists*$(\langle x\rangle, \langle y\rangle, W)$.

Figure 1: Unification of WAM Objects

b) U has an environment iff W has an environment; if so, then $deallocate(U)$ is an extension of $deallocate(W)$ by θ

c) U has a choice point iff W has a choice point; if so, then $backtrack(U) \approx backtrack(W)$

d) for every object ξ of W, ξ is an object of U and $[\xi]_U = [\xi]_W\theta$.

As an immediate consequence of this definition, we have

Lemma 3.12 *Let W, U, and V be directed WAM states and let σ and τ be substitutions. If U is an extension of W by σ and V is an extension of U by τ, then V is an extension of W by $\sigma\tau$.*

Lemma 3.13 *Let W be a directed WAM state, let x be an object of W, and let v be an unbound variable of W. Assume that v does not occur in x, and that if x is a variable, then v does not precede x. Then $bind(v, x, W)$ is an extension of W by $\{([v]_W, [x]_W)\}$.*

Proof: We show by induction on $W(\mathbf{e})$ that Definition 3.28 holds for $U = bind(v, x, W)$ and $\theta = \{([v]_W, [x]_W)\}$:

a) is trivially satisfied.

b) is trivial if W has no environment. Otherwise, let $W' = deallocate(W)$ and $U' = deallocate(U)$. If v is a stable object of W, then $U' = bind(v, x, W')$ and the statement follows by induction. If not, then $U' \equiv W'$, $[\xi]_{U'} = [\xi]_{W'} = [\xi]_{W'}\theta$ for each object ξ of W', and the statement is again trivial.

c) follows from Lemma 3.8.

d) follows from Lemma 3.9 and induction on $count(\xi, W)$.$\square$

Finally, our specification of $unify$ is as follows:

Lemma 3.14 *Let x and y be objects of a directed WAM state W and let $U = unify(x, y, W)$. If $[x]_W$ and $[y]_W$ are unifiable terms, then they have a mgu θ such that U is an extension of W by θ. If not, then $U \approx backtrack(W)$.*

Proof: For any list $p = \langle \xi_k, \ldots, \xi_1 \rangle$ of objects of a WAM state W, $[p]_W$ will denote the list of terms $\langle [\xi_k]_W, \ldots, [\xi_1]_W \rangle$. Let $l = \langle u_n, \ldots, u_1 \rangle$ and $m = \langle v_n, \ldots, v_1 \rangle$ be lists of objects of W and let $U = unify\text{-}lists(l, m, W)$. We shall show that if $[l]_W$ and $[m]_W$ are not unifiable, then $U \approx backtrack(W)$, while if $[l]_W\sigma = [m]_W\sigma$ for some substitution σ, then there exist substitutions θ and τ such that U is an extension of W by θ, $[l]_W\theta = [m]_W\theta$, and $\sigma = \theta\tau$.

If either $l = \texttt{NIL}$ or W has no unbound variables, then $U = W$ and the claim is trivially true with $\theta = \emptyset$ and $\sigma = \tau$. We proceed by induction, assuming that $n > 0$ and that the claim is true if either W is replaced by a state with fewer unbound variables than W, or W is held constant and $l = \langle u_n, \ldots, u_1\rangle$ is replaced by a list $l' = \langle u'_{n'}, \ldots, u'_1\rangle$ such that

$$\sum_{i=1}^{n'} count(u'_i, W) < \sum_{i=1}^{n} count(u_i, W).$$

Suppose $n > 0$ and let $x = deref(u_n, W)$ and $y = deref(v_n, W)$. If $x = y$, then $U = unify\text{-}lists(cdr(l), cdr(m), W)$ and $[u_n]_W = [x]_W = [y]_W = [v_n]_W$. Thus, for any σ, $[l]_W\sigma = [m]_W\sigma$ iff $cdr([l]_W)\sigma = cdr([m]_W)\sigma$, and the claim follows from the inductive hypothesis.

Next, suppose that x and y are structures with the same functor. Let $l' = append(args(x, W), cdr(l))$ and $m' = append(args(y, W), cdr(m))$. Then $U = unify\text{-}lists(l', m', W)$ and for any σ, $[l]_W\sigma = [m]_W\sigma$ iff $[l']_W\sigma = [m']_W\sigma$. The claim again follows by induction.

In the remaining case, if neither x nor y is a variable, or if one is a variable that occurs in the other, then $[x]_W$ and $[y]_W$ are not unifiable, hence neither are $[l]_W$ and $[m]_W$, and $U = backtrack(W)$. We may assume, therefore, that either x or y is a variable that does not occur in the other. Moreover, without loss of generality, we may assume that x is a variable that does not occur in y and that if y is also a variable, then y precedes x.

In this case, $U = unify\text{-}lists(l', m', W')$, where $l' = cdr(l)$, $m' = cdr(m)$, and $W' = bind(x, y, W)$. By Lemma 3.13, W' is an extension of W by $\rho = \{([x]_W, [y]_W)\}$. If $[l]_W$ and $[m]_W$ are not unifiable, then neither are $[l']_{W'} = cdr([l]_W)\rho$ and $[m']_{W'} = cdr([m]_W)\rho$, hence $U \approx backtrack(W') \approx backtrack(W)$. Suppose $[l]_W\sigma = [m]_W\sigma$. Let $\sigma' = \sigma - \{([x]_W, [x]_W\sigma)\}$. If z is any variable symbol other than $[x]_W$, then $z\rho\sigma' = z\sigma' = z\sigma$, while $[x]_W\rho\sigma' = [y]_W\sigma' = [y]_W\sigma = [x]_W\sigma$. Thus, $\rho\sigma' = \sigma$ and $[l']_{W'}\sigma' = cdr([l]_W)\rho\sigma' = cdr([l]_W)\sigma = cdr([m]_W)\sigma = cdr([m]_W)\rho\sigma' = [m']_{W'}\sigma'$. By inductive hypothesis, there exist θ' and τ such that U is an extension of W' by θ', $[l']_{W'}\theta' = [m']_{W'}\theta'$, and $\sigma' = \theta'\tau$. Let $\theta = \rho\theta'$. Then

$$\sigma = \rho\sigma' = \rho\theta'\tau = \theta\tau,$$

$$
\begin{aligned}
[l]_W\theta &= (car([l]_W)\rho\theta', cdr([l]_W)\rho\theta') = ([x]_W\rho\theta', [l']_{W'})\theta') \\
&= ([y]_W\rho\theta', [m']_{W'})\theta') = (car([m]_W)\rho\theta', cdr([m]_W)\rho\theta') = [m]_W\theta,
\end{aligned}
$$

and by Lemma 3.12, U is an extension of W by θ.□

3.7 WAM Instructions

We assume that $\mathbf{O}$ is a set of order 22:

$$
\begin{aligned}
\mathbf{O} = \{&\texttt{TRY-ME-ELSE}, \texttt{RETRY-ME-ELSE}, \texttt{TRUST-ME-ELSE-FAIL}, \\
&\texttt{BACKTRACK}, \texttt{JUMP}, \texttt{PROCEED}, \texttt{ALLOCATE}, \texttt{DEALLOCATE}, \texttt{CALL}, \\
&\texttt{EXECUTE}, \texttt{GET-VALUE}, \texttt{GET-VARIABLE}, \texttt{GET-CONSTANT}, \\
&\texttt{GET-STRUCTURE}, \texttt{PUT-VALUE}, \texttt{PUT-VARIABLE}, \texttt{PUT-CONSTANT}, \\
&\texttt{PUT-STRUCTURE}, \texttt{PUT-UNSAFE-VALUE}, \texttt{UNIFY-VALUE}, \\
&\texttt{UNIFY-VARIABLE}, \texttt{UNIFY-CONSTANT}\},
\end{aligned}
$$

each element of which is called a *WAM opcode*. We further assume that associated with each opcode ω is a function T_ω, and for convenience, we shall give T_ω the same name as ω, for each $\omega \in \mathbf{O}$. Thus, the first eight opcodes listed above are associated with the functions *try-me-else*, *retry-me-else*, *trust-me-else-fail*, *backtrack*, *jump*, *proceed*, *allocate*, and *deallocate*, all of which have been previously defined. The functions associated with the remaining opcodes will be defined later in this section.

For $\omega \in \mathbf{O}$, the function T_ω may have any positive arity. Its last argument is expected to be a WAM state, and the number of remaining arguments is called the *arity* of ω. Thus, if T_ω is a function of arity n, then the arity of ω is $n - 1$. The first $n - 1$ arguments of T_ω may be natural numbers, constants, predicate and function symbols, or elements of the sets $\mathbf{X}$ and $\mathbf{Y}$. These are assumed to be countably infinite sets, $\mathbf{X} = \{X_1, X_2, \ldots\}$ and $\mathbf{Y} = \{Y_1, Y_2, \ldots\}$, of objects called *temporary symbols* and *permanent symbols*, respectively. Temporary and permanent symbols refer to temporary and permanent variables, respectively. We define the following mapping:

Definition 3.29 *Let W be a WAM state.*

a) For each $X_i \in \mathbf{X}$, $|X_i|_W = A_i$.

b) For each $Y_i \in \mathbf{Y}$, $|Y_i|_W = pv(i, W)$.

c) For each $z \in \mathbf{X} \cup \mathbf{Y}$, $[z]_W = [W(|z|_W)]_W$.

Definition 3.30 *A* **WAM instruction** *is a list* $\langle \omega, x_1, \ldots, x_n \rangle$*, where* $\omega \in \mathbf{O}$*,* $n = arity(\omega)$*, and* $x_i \in \mathbf{N} \cup \mathbf{K} \cup \mathbf{P} \cup \mathbf{F} \cup \mathbf{X} \cup \mathbf{Y}$ *for* $1 \leq i \leq n$*. The* x_i *are called the* **arguments** *of the instruction.*

An instruction $I = \langle \omega, x_1, \ldots, x_n \rangle$ is encoded in the WAM memory by storing its members in $n+1$ consecutive cells in the code area. Suppose that with respect to a WAM state W, I is stored at address k, i.e., $W(C_k) = \omega$ and $W(C_{k+i}) = x_i$, for $i = 1, \ldots, n$. If $W(\mathbf{p}) = k$, then I is the instruction to be executed in this state. After the program counter is advanced by $n+1$, resulting in a state W', the function T_ω is applied to the $n+1$ arguments $x_1, \ldots, x_n, W'$. The function that performs this execution is defined as follows:

Definition 3.31 *Let* W *be a WAM state. Suppose that* $\omega = read(0, W) \in \mathbf{O}$ *with* $arity(\omega) = n$*. Let* $x_i = read(i, W)$ *for* $i = 1, \ldots, n$*, and let* $W' = advance(n+1, W)$*. Then*

$$step(W) = T_\omega(x_1 \ldots, x_n, W').$$

Definition 3.32 *If* $W_0, \ldots, W_n$ *is a sequence of WAM states such that* $W_i = step(W_{i-1})$ *for* $i = 1, \ldots, n$*, then*

$$W_0 \rightarrow W_n.$$

Definition 3.33 *A WAM state* W **fails** *if* $W \rightarrow W'$*, where either*

a) $W' \approx backtrack(W)$*, or*

b) W' *is not a success state and* $step(W')$ *is undefined.*

Definition 3.34 *Let* W *be a WAM state and let* $k \in \mathbf{N}$*. Let*

$$L = \langle W(C_k), \ldots, W(C_n) \rangle,$$

where $k \leq n$*. Then* k **points to** L *with respect to* W*. If* $k = 1$*, then* L *is* **loaded** *w.r.t.* W*.*

It is useful to reformulate Definition 3.31 in terms of Definition 3.34:

Lemma 3.15 *Let W be a WAM state and let $I = \langle \omega, x_1, \ldots, x_n \rangle$ be a WAM instruction. If $W(\mathbf{p})$ points to append(I,L) for some list L, then*

$$step(W) = T_\omega(x_1 \ldots, x_n, W'),$$

where $W' = jump(j, W)$ and j points to L w.r.t. W.

As we shall see in Section 4.4, if the program P is compiled and loaded with respect to a state W, then the contents of the first $2r$ cells of the code area, $\langle W(C_1), \ldots, W(C_{2r}) \rangle$, is the list $\langle q_1, a_1, \ldots, q_r, a_r \rangle$, where $\{q_1, \ldots, q_r\}$ is the set of all predicate symbols that occur in P and a_i is the address in the code area where the compiled procedure for q_i is stored. Thus, the following function may be used to compute the address of the procedure for a given predicate:

Definition 3.35 *Let W be a WAM state, let q be a predicate symbol, and let n be the least natural number for which $W(C_n) = q$. Then*

$$procedure\text{-}address(W, q) = W(C_{n+1}).$$

The instructions `EXECUTE` and `CALL` are both used for jumping to a WAM procedure:

Definition 3.36 *Let q be a predicate symbol defined in W.*

a) $execute(q, W) = jump(procedure\text{-}address(q, W), set\text{-}arity(q, W))$.

b) $call(q, W) = execute(q, set\text{-}return\text{-}address(W))$.

The four `GET-` instructions are used to unify the contents of a temporary variable with some other WAM object:

Definition 3.37 *If W is a WAM state, $x \in \mathbf{X}$, $z \in \mathbf{X} \cup \mathbf{Y}$, and $c \in \mathbf{K}$, then*

a) $get\text{-}value(z, x, W) = unify(W(|z|_W), W(|x|_W), W)$

b) $get\text{-}variable(z, x, W) = bind(|z|_W, W(|x|_W), W)$

c) $get\text{-}constant(c, x, W) = unify(c, W(|x|_W), W)$

In order to define `GET-STRUCTURE`, we need an operation that creates a structure by pushing a function symbol and a block of unbound variables onto the heap and setting the structure pointer to the address of the new structure:

Definition 3.38 *Let W be a WAM state, let f be a function symbol of arity n, and let $k = W(\mathbf{h}) + 1$. Then push-structure(f, W) is the state*

$$W[H_k \leftarrow f, H_{k+1} \leftarrow H_{k+1}, \ldots, H_{k+n} \leftarrow H_{k+n}, \mathbf{h} \leftarrow k+n, \mathbf{s} \leftarrow k].$$

We shall also require an operation that sets the structure pointer to the address of the first argument of an already existing structure:

Definition 3.39 *If $x = H_k$ is a structure of a WAM state W, then*

$$\textit{set-structure-pointer}(x, W) = W[\mathbf{s} \leftarrow k].$$

Definition 3.40 *Let W be a WAM state, let $x \in \mathbf{X}$, $f \in \mathbf{F}$, and let $dx = \textit{deref}(W(|x|_W), W)$. Then $\textit{get-structure}(f, x, W)$ is the state*

a) $\textit{unify}(dx, H_{W(\mathbf{h})+1}, \textit{push-structure}(f, W))$, if dx is a variable of W,

b) $\textit{set-structure-pointer}(dx, W)$, if dx is a structure of W with functor f,

c) backtrack(W), otherwise.

The `PUT-` instructions are used to store WAM objects into temporary variables:

Definition 3.41 *If W is a WAM state, $x \in \mathbf{X}$, $c \in \mathbf{K}$, $z \in \mathbf{X} \cup \mathbf{Y}$, $y \in \mathbf{Y}$, and $dy = \textit{deref}(|y|_W, W)$, then*

a) $\textit{put-value}(z, x, W) = \textit{bind}(|x|_W, W(|z|_W), W)$

b) $$\textit{put-variable}(z, x, W) = \begin{cases} \textit{bind}(|x|_W, |z|_W, W) & \textit{if } z \in \mathbf{Y} \\ \textit{stabilize}(|x|_W, |z|_W, W) & \textit{if } z \in \mathbf{X} \end{cases}$$

c) $\textit{put-constant}(c, x, W) = \textit{bind}(|x|_W, c, W)$

d) $\textit{put-structure}(f, x, W) = \textit{bind}(|x|_W, H_{W(\mathbf{h})+1}, \textit{push-structure}(f, W))$

e) $$\textit{put-unsafe-value}(y, x, W) = \begin{cases} \textit{stabilize}(|x|_W, dy, W) & \textit{if } dy \textit{ is unstable} \\ \textit{bind}(|x|_W, dy, W) & \textit{otherwise} \end{cases}$$

The `UNIFY-` instructions, which are only called after either `GET-STRUCTURE` or `PUT-STRUCTURE`, attempt to unify heap variables with other WAM objects:

Definition 3.42 *If W is a WAM state, $z \in \mathbf{X} \cup \mathbf{Y}$, and $c \in \mathbf{K}$, then*

a) $\textit{unify-variable}(z, W) = \textit{bind}(|z|_W, H_{W(\mathbf{s})+1}, \textit{next-arg}(W))$

b) $\textit{unify-value}(z, W) = \textit{unify}(W(|z|_W), H_{W(\mathbf{s})+1}, \textit{next-arg}(W))$

c) $\textit{unify-constant}(c, W) = \textit{unify}(c, H_{W(\mathbf{s})+1}, \textit{next-arg}(W))$

4 Prolog on the WAM

4.1 Specification of the Compiler

We shall define two functions, *compile-program* and *compile-goal*, such that for any program p and any goal clause g, *compile-program(p)* and *compile-goal(g)* are lists of elements of $\mathbf{U}$. Program execution is defined in terms of these two functions as follows:

Definition 4.1 *Let p be a program and let g be a goal clause with $vars(g) = \langle v_1, \ldots, v_n \rangle$. Let W_0 be a WAM state such that*

a) $W_0(\mathbf{b}) = W_0(\mathbf{e}) = W_0(\mathbf{c}) = W_0(\mathbf{t}) = 0$,

b) $W_0(\mathbf{a}) = W_0(\mathbf{h}) = n$,

c) $W_0(A_i) = W_0(H_i) = H_i$, for $i = 1, \ldots, n$,

d) compile-program(p) is loaded w.r.t. W_0, and

e) $W_0(\mathbf{p})$ points to compile-goal(g) w.r.t. W_0.

Let $\langle W_k, \ldots, W_1 \rangle$ be a list of success states such that $W_0 \rightarrow W_1$ and

$$backtrack(W_i) \rightarrow W_{i+1}$$

for $i = 1, \ldots, k-1$. For $i = 1, \ldots, k$, let

$$\sigma_i = \{(v_j, [H_j]_{W_i}) | 1 \leq j \leq n\}$$

and let $\bar{\sigma}_i$ denote the set of substitutions for g that are equivalent to σ_i. Then $\langle \bar{\sigma}_k, \ldots, \bar{\sigma}_1 \rangle$ is called a **WAM execution** *of p and g.*

Once we have defined *compile-program* and *compile-goal*, our goal will be the proof of

Theorem 4.1 *Let Σ be a list of equivalence classes of substitutions for G. Then Σ is a WAM execution for P and G if and only if Σ is an interpretation of P and G.*

In the sequel, W_0 will denote a fixed WAM state that satisfies Definition 4.1 with respect to our fixed program P and goal clause G. For any WAM state W, we shall assume that $W(C_i) = W_0(C_i)$ for all $i \geq 1$. In this context, the statements *k points to L* and *L is loaded* will be unambiguous. If $vars(G)$ is the list $\langle v_1, \ldots, v_n \rangle$, then σ_W will denote the substitution $\{(v_1, [H_1]_W), \ldots, (v_n, [H_n]_W)\}$.

4.2 Symbol Tables

A variable symbol occurring in a program clause corresponds at run-time either to a permanent variable or to a temporary variable of the WAM, according to whether or not it must be saved across a subroutine call:

Definition 4.2 *Let v be a variable symbol that occurs in a program clause C. If v occurs in two distinct goal literals of C, or if v occurs in both the head literal and some goal literal other than the first, then v is* **permanent** *with respect to C. Otherwise, v is* **temporary** *with respect to C. The set of all variable symbols that are permanent w.r.t. C is denoted $perms(C)$.*

The permanent variable symbols of a clause are further classified as *safe* or *unsafe*:

Definition 4.3 *Let ℓ be the last goal literal of a program clause C and let $v \in perms(C)$. Then v is* **safe** *with respect to C if any of the following holds:*

a) v does not occur in ℓ;

b) v occurs in head(C);

c) v occurs in a compound term occurring in $C - \langle \ell \rangle$;

d) the first occurrence of v in ℓ is in a compound term.

Otherwise, v is **unsafe** *w.r.t. C. The set of all unsafe variable symbols w.r.t. C is denoted* **uns**(*C*).

The significance of an unsafe variable symbol is that the value of its corresponding run-time variable may be an unstable object at the time that it is passed as an argument to the procedure call corresponding to the last goal literal of a clause. Since the current environment is deallocated immediately before passing control to this procedure, special precautions (as described in Section 4.3) are necessary in the presence of unsafe variable symbols in order to avoid dangling references.

The correspondence between variable symbols and permanent and temporary variables is represented as an object of the following type:

Definition 4.4 *A* **symbol table** *is a finite subset θ of $\mathbf{V} \times (\mathbf{X} \cup \mathbf{Y})$ such that if (x_1, z_1) and (x_2, z_2) are distinct elements of θ, then $x_1 \neq x_2$ and $z_1 \neq z_2$.*

Note that in the context of a WAM state, a symbol table determines a substitution:

Definition 4.5 *If W is a WAM state and $\theta = \{(v_1, z_1), \ldots, (v_s, z_s)\}$ is a symbol table, then*

$$[\theta]_W = \{(v_1, [z_1]_W), \ldots, (v_s, [z_s]_W)\}.$$

Each literal ℓ of a program clause C is compiled in the context of a symbol table b. Of the variable symbols that occur in ℓ, the ones that occur in b are precisely those that also occur in some literal of C that precedes ℓ. As new variable symbols are encountered during the compilation of ℓ, a new symbol table b' is constructed as an extension of b. Thus, if a variable symbol v has its first occurrence in ℓ, then v is assigned to a permanent or temporary symbol and this assignment is recorded in b'.

If $v \in perms(C)$, then the permanent symbol of lowest index that does not already occur in (the partially constructed) b' is selected. The selection of a temporary symbol, in the case $v \notin perms(C)$, is in general restricted by two other parameters: a list r of temporary symbols that are already in use for some other purpose, and a natural number n, which represents a strict lower bound on the index of a selected symbol. The latter restriction is imposed in order to avoid allocating temporary variables that are reserved for procedure arguments.

The following functions are used in the construction of b':

Definition 4.6 *Let b be a symbol table, let $n \in \mathbf{N}$, and let r be a list of temporary symbols.*

a) new-perm$(b) = Y_k$, k being the least i such that Y_i does not occur in b;

b) new-temp$(b, r, n) = X_k$, k being the least i such that $i > n$ and X_i does not occur in either r or b.

The resulting symbol table is described in terms of

Definition 4.7 *Let b be a symbol table, let $p \subset \mathbf{V}$ be a set of m variable symbols, and let $n \in \mathbf{N}$. Then b* **respects** *p and n if for each $(x, z) \in b$, either*

a) $x \in p$ and $z = Y_i$ for some $i \leq m$, or

b) $x \notin p$ and $z = X_j$ for some $j > n$.

The unstable variables of a WAM state corresponding to permanent symbols that have not yet been assigned to variable symbols require special attention:

Definition 4.8 *Let b be a symbol table, let W be a WAM state, and let $\xi = pv(k, W)$ be an unstable variable of W. If Y_k does not occur in b, then ξ is a* **reserved variable** *of W with respect to b.*

Definition 4.9 *Let b be a symbol table and let W be a WAM state. A permanent variable ξ of W is* **free** *with respect to b if ξ is unbound, no variable of W is bound to ξ, and for all X_j occurring in b, $W(A_j) \neq \xi$. If p is a set of m variable symbols, then W is* **compatible** *with b and p if $W(|z|_W)$ is an object of W for all $(x, z) \in b$, and if $m > 0$, then W has an environment of order m and each reserved variable of W w.r.t. b is free w.r.t. b.*

4.3 Compiling Literals

This section describes three functions on which our definitions of *compile-program* and *compile-goal* will depend: *compile-head-literal*, *compile-goal-literal*, and *extend-table*. We begin by presenting specifications for these functions—that is, we state three lemmas that together characterize their desired behavior. We then give definitions for the three functions and prove that they satisfy these specifications.

The function *extend-table* returns the symbol table b' that is constructed during the compilation of a (head or goal) literal of a program clause. Its arguments are the list t of arguments of the literal, the set p of permanent variable symbols of the clause, the symbol table b in the context of which the literal is compiled, and a natural number n restricting the choice of temporary symbols, as previously described. For a goal literal, n is the number of its arguments (i.e., the length of t); for a head literal, n is the number of arguments of the leading goal literal (or $n = 0$ if the body is `NIL`). The behavior of *extend-table* is specified by

Lemma 4.1 *Let t be a list of terms, $p \subset \mathbf{V}$, $n \in \mathbf{N}$, b a symbol table that respects p and n, and $b' = extend\text{-}table(t, b, p, n)$. Then b' is a symbol table that respects p and n, $b \subseteq b'$, and the variable symbols that occur in b' are all of those that occur in either b or t.*

The function *compile-head-literal* is designed to generate code that attempts to match the arguments of the head of a clause with the actual arguments of a procedure call, which have been stored in argument registers. It takes three arguments: the argument list t of the head literal, the set p of permanent variable symbols of the clause, and the number n of arguments of the first goal literal of the clause. (The last argument is used to ensure that the first n argument registers are reserved for the arguments of the first subroutine call.) The properties of this function that we shall require are collected in

Lemma 4.2 *Let $t = \langle t_1, \ldots, t_k \rangle$ be a list of terms in which no element of range(N) occurs. Let $n \in \mathbf{N}$, $p \subset \mathbf{V}$, and $b = \mathit{extend\text{-}table}(t, \emptyset, p, n)$. Let W be a directed WAM state compatible with $\emptyset$ and p such that $W(\mathbf{p})$ points to*

$$\mathit{append}(\mathit{compile\text{-}head\text{-}literal}(t, p, n), d).$$

Assume that $W(A_i)$ is an object of W for $i = 1, \ldots, k$ and let

$$u = \langle [A_1]_W, \ldots, [A_k]_W \rangle.$$

If t and u are not unifiable, then W fails. Otherwise $W \to U$, where

a) $U(\mathbf{p})$ points to d,

b) U is an extension of W by some substitution θ such that $[b]_U \subseteq \theta$ and θ is a mgu for t and u,

c) if ξ is a free variable of W w.r.t. $\emptyset$ and $\xi \neq |z|_U$ for all $(x, z) \in b$, then ξ is a free variable of U w.r.t. b; in particular, U is compatible with b and p. Suppose further that $W(A_i)$ is a stable object of W for $i = 1, \ldots, k$. Then for each $(x, z) \in b$, $|z|_U$ is a stable object of U.

The code for each goal literal of a clause is generated by *compile-goal-literal*, which takes four arguments: the list t of arguments of the literal, the set p of permanent variable symbols of the clause, the symbol table b in the context of which the compilation occurs, and a set $u \subseteq p$. In the case of the final goal literal of a clause, u is the set of unsafe variable symbols of the clause; otherwise, $u = \emptyset$. The resulting code has the effect of storing objects that represent the arguments of the literal into argument registers. The precise specification of *compile-goal-literal* is

Lemma 4.3 *Let $t = \langle t_1, \ldots, t_k \rangle$ be a list of terms, $p \subset \mathbf{V}$, b a symbol table that respects p and n, and $b' = extend\text{-}table(t, b, p, n)$. Let $v \subseteq p$ such that for all $x \in v$, $(x, z) \in b$ for some $z \in \mathbf{Y}$, and let W be a directed WAM state compatible with b and p such that $W(\mathbf{p})$ points to*

$$append(compile\text{-}goal\text{-}literal(t, b, p, v), c).$$

Then $W \to U$, where

a) $U(\mathbf{p})$ points to c,

b) U is an extension of W by some renaming substitution θ such that $[b' - b]_U \subseteq \theta$ and for each $(x, y) \in \theta$ there is a variable ξ of U such that $N(\xi) = y$ and if ξ is a variable of W, then ξ is reserved w.r.t. b,

c) if ξ is a free variable of W w.r.t. b and $\xi \neq |z|_U$ for all $(x, z) \in b'$, then ξ is a free variable of U w.r.t. b'; in particular, U is compatible with b' and p,

d) $[A_i]_U = t_i[b']_U$ for $i = 1, \ldots, k$,

e) for each $(x, z) \in b' - b$, if x occurs in a compound term that occurs in t, then $U(|z|_U)$ is stable.

Suppose further that for each $x \in p - v$ that occurs in t, either i) there exists a pair $(x, z) \in b$ such that $W(|z|_W)$ is stable, or ii) the first occurrence of x in t is in a compound term. Then for $j = 1, \ldots, k$, $U(|X_{i_j}|_U)$ is stable.

The functions *compile-head-literal*, *compile-goal-literal*, and *extend-table* are defined in Figures 2, 3, and 4 in terms of the auxiliary functions *compile-literal*, *compile-term*, and *extend-table-aux*. In our design of these definitions, simplicity was a greater consideration than efficiency. However, since our proof of Theorem 4.1 depends on these definitions only insofar as they satisfy the three lemmas stated above, they may be optimized in any way that preserves the truth of these lemmas.

Note that the functions *compile-literal* and *compile-term* serve to generate the code for both head and goal literals. The main difference between these two cases is that the code for a head literal *reads* objects from argument registers by means of the `GET-` instructions, whereas the code for a goal literal *writes* objects into argument registers by means of the corresponding `PUT-` instructions. The value of the argument h to *compile-literal* and *compile-term* is expected to be either `GET-` or `PUT-`, according to the type of literal being compiled. The function *concat* is assumed to return the WAM opcode that corresponds to the concatenation of its two arguments, e.g., $concat(\texttt{PUT-}, \texttt{VALUE}) = \texttt{PUT-VALUE}$.

The three auxiliary functions take another argument, r, which is a list of temporary symbols corresponding to the argument registers from which arguments are to be read or into which arguments are to be written. The functions *compile-term* and *extend-table* also take an argument a, representing the arguments of a compound term being compiled.

Lemmas 4.1, 4.2, and 4.3 will be derived as consequences of three more general statements about the functions *compile-term* and *extend-table-aux*. These will all be proved according to the same induction scheme, based on the structure of the expression (a, t), where a and t are lists of terms. The inductive hypothesis is that the proposition to be proved is true if a and t are replaced by $\tilde{a}$ and $\tilde{t}$, where either $complexity((\tilde{a}, \tilde{t})) < complexity((a, t))$, or $complexity((\tilde{a}, \tilde{t})) = complexity((a, t))$ and $length(\tilde{a}) < length(a)$.

Using this induction scheme, the proof of the following is a straightforward case analysis based on the definition of *extend-table-aux*. Note that Lemma 4.1 is the special case where $a = \texttt{NIL}$ and $r = \langle X_1, \ldots, X_{length(t)} \rangle$:

Lemma 4.4 *Let a and t be lists of terms, r a list of temporary symbols of the same length as t, $p \subset \mathbf{V}$, $n \in \mathbf{N}$, b a symbol table that respects p and n, and $b' = extend\text{-}table(a, t, r, b, p, n)$. Then b' is a symbol table that respects p and n, such that $b \subseteq b'$ and the variable symbols that occur in b' are all of those that occur in either b or t.*

Lemma 4.2 is a special case of the following, with $\ell = 0$, $r = \langle X_1, \ldots, X_k \rangle$, and $b = \emptyset$:

Lemma 4.5 *Let $a = \langle a_1, \ldots, a_\ell \rangle$ and $t = \langle t_1, \ldots, t_k \rangle$ be lists of terms in which no element of $range(N)$ occurs. Let $r = \langle X_{i_1}, \ldots, X_{i_k} \rangle$ be a list of temporary symbols, $n \in \mathbf{N}$, and $p \subset \mathbf{V}$. Let b be a symbol table that respects p and n such that no member of r occurs in b, and let*

$$b' = extend\text{-}table\text{-}aux(a, t, r, b, p, n).$$

Let W be a directed WAM state compatible with b and p such that $W(\mathbf{p})$ points to

$$append(compile\text{-}term(\texttt{GET-}, a, t, r, b, p, \emptyset, n), d).$$

```
Function compile-literal(h,t,r,b,p,u,n);
begin
  if t = NIL then return NIL
  else if car(t) is a constant then
    i ← ⟨concat(h, CONSTANT), car(t), car(r)⟩
  else if car(t) is a compound term then
    begin
      if car(r) = X_j with j > n
        then i ← ⟨GET-STRUCTURE, funct(car(t)), car(r)⟩
        else i ← ⟨concat(h, STRUCTURE), funct(car(t)), car(r)⟩;
      return append(i, compile-term(h, args(car(t)), cdr(t), cdr(r), b, p, u, n))
    end
  else if car(t) ∈ u and (car(t), z) ∈ b then *{unsafe variable}*
    i ← ⟨PUT-UNSAFE-VALUE, z, car(r)⟩
  else if (car(t), z) ∈ b then *{previously encountered variable}*
    i ← ⟨concat(h, VALUE), z, car(r)⟩
  else *{first occurrence of variable}*
    begin
      if car(t) ∈ p
        then z ← new-perm(b)
        else z ← new-temp(b, r, n);
      b ← {(car(t), z)} ∪ b;
      i ← ⟨concat(h, VARIABLE), z, car(r)⟩
    end;
  return append(i,compile-literal(h,cdr(t),cdr(r),b,p,u − {car(t)},n))
end.
```

Figure 2: Compiling a Literal

```
Function compile-term(h,a,t,r,b,p,u,n);
begin
  if a = NIL then return compile-literal(h, t, r, b, p, u, n)
  else if car(a) is a constant then
    i ← ⟨UNIFY-CONSTANT, car(a)⟩
  else if car(a) is a compound term then
    begin
      t ← (car(a), t); r ← (new-temp(b, r, n), r);
      i ← ⟨UNIFY-VARIABLE, car(r)⟩
    end
  else if (car(a), z) ∈ b then i ← ⟨UNIFY-VALUE, z⟩
  else begin
         if car(a) ∈ p
           then z ← new-perm(b)
           else z ← new-temp(b, r, n);
         b ← {(car(a), z)} ∪ b;
         i ← ⟨UNIFY-VARIABLE, z)⟩
       end;
  return append(i, compile-term(h, cdr(a), t, r, b, p, u, n)
end.

Function compile-head-literal(t,p,n);
return compile-literal(GET-, t, ⟨X_1, ..., X_length(t)⟩, ∅, p, ∅, n).

Function compile-goal-literal(t,b,p,v);
return compile-literal(PUT-, t, ⟨X_1, ..., X_length(t)⟩, b, p, v, length(t)).
```

Figure 3: Compiling a Literal

```
Function extend-table-aux(a,t,r,b,p,n);
begin
  if a = NIL then
    begin
      if t = NIL then return b
      else if car(t) is a compound term then
        return extend-table-aux(args(car(t)), cdr(t), cdr(r), b, p, n)
      else if car(t) is a variable symbol occurring in b or a constant then
        return extend-table-aux(NIL, cdr(t), cdr(r), b, p, n)
      else begin
             if car(t) ∈ p
               then b ← {(car(t), new-perm(b))} ∪ b
               else b ← {(car(t), new-temp(b, r, n))} ∪ b;
             return extend-table-aux(NIL, cdr(t), cdr(r), b, p, n)
           end
    end
  else if car(a) is a variable symbol not occurring in b then
    if car(a) ∈ p
      then b ← {(car(a), new-perm(b))} ∪ b
      else b ← {(car(a), new-temp(b, r, n))} ∪ b
  else if car(a) is a compound term then
    begin
      t ← (car(a), t); r ← (new-temp(b, r, n), r)
    end;
  return extend-table-aux(cdr(a),t,r,b,p,n)
end.

Function extend-table(t,b,p,n);
return extend-table-aux(NIL, t, ⟨X_1, ..., X_length(t)⟩, b, p, n).
```

Figure 4: Building a symbol table

Assume that $W(A_i)$ is an object of W for $i = 1, \ldots, k$ and that $\xi_i = H_{W(\mathbf{s})+i}$ is a variable of W for $i = 1, \ldots, \ell$. Let

$$L = \langle [\xi_1]_W, \ldots, [\xi_\ell]_W, [X_{i_1}]_W, \ldots, [X_{i_k}]_W \rangle$$

and $M = append(a, t)[b]_W$.

If L and M are not unifiable, then W fails. Otherwise, $W \rightarrow U$, where

a) $U(\mathbf{p})$ points to d,

b) U is an extension of W by some substitution θ such that $[b' - b]_U \subseteq \theta$ and θ is a mgu for L and M,

c) if ξ is a free variable of W w.r.t. b and $\xi \neq |z|_U$ for all $(x, z) \in b'$, then ξ is a free variable of U w.r.t. b'; in particular, U is compatible with b' and p.

Suppose further that $W(|X_{i_j}|_W)$ is stable for $j = 1, \ldots, k$. Then for each $(x, z) \in b' - b$, $U(|z|_U)$ is stable.

Proof: The subcases considered below are determined by the definitions of *compile-literal* and *compile-term*. In each subcase, the statement of the lemma is derived from some instance of the inductive hypothesis, obtained by replacing W by $\tilde{W} = step(W)$, and a, t, r, b, L, and M by some $\tilde{a}$, $\tilde{t}$, $\tilde{r}$, $\tilde{b}$, $\tilde{L}$, and $\tilde{M}$, respectively.

Case 1: $a \neq$ `NIL`

This has four subcases, determined by the definition of *compile-term*. In each of these, $\tilde{a} = cdr(a)$:

subcase 1.1: a_1 is a constant

Here, $\tilde{W} = unify\text{-}constant(a_1, advance(2, W))$. If $[\xi_1]_W$ and $a_1[b]_W = a_1$ are not unifiable, then neither are L and M. In this case, $\tilde{W} = backtrack(W)$ and the proposition holds true. Assuming they are unifiable, $\tilde{W}$ is an extension of W by

$$\tau = \begin{cases} \{([\xi_1]_W, a_1)\}, \text{ if } [\xi_1]_W \text{ is a variable symbol} \\ \emptyset, \text{ if } [\xi_1]_W = a_1. \end{cases}$$

Let $\tilde{t} = t$, $\tilde{r} = r$, and $\tilde{b} = b$. Then

$$\tilde{L} = \langle [\xi_2]_{\tilde{W}}, \ldots, [\xi_\ell]_{\tilde{W}}, [X_{i_1}]_{\tilde{W}}, \ldots, [X_{i_k}]_{\tilde{W}} \rangle = cdr(L)\tau$$

and $\tilde{M} = cdr(M)\tau$. If L and M are not unifiable, then neither are $\tilde{L}$ and $\tilde{M}$, and hence $\tilde{W}$ fails. If L and M are unifiable, then so are $\tilde{L}$ and $\tilde{M}$, which must then have a mgu $\tilde{\theta}$ such that $\tilde{W} \to U$, where U is an extension of $\tilde{W}$ by $\tilde{\theta}$. But then U is an extension of W by $\theta = \tau\tilde{\theta}$ and θ is a mgu for L and M.

subcase 1.2: a_1 is a compound term

Here, $\tilde{b} = b$, $\tilde{t} = (a_1, t)$, $\tilde{r} = (X_j, r)$, where $X_j = \mathit{new\text{-}temp}(b, r, n)$, $\tilde{W} = \mathit{unify\text{-}variable}(X_j, \mathit{advance}(2, W))$,

$$\tilde{L} = \langle [\xi_2]_{\tilde{W}}, \ldots, [\xi_\ell]_{\tilde{W}}, [X_j]_{\tilde{W}}, [X_{i_1}]_{\tilde{W}}, \ldots, [X_{i_k}]_{\tilde{W}} \rangle,$$

and

$$\tilde{M} = \langle a_2, \ldots, a_\ell, a_1, t_1, \ldots, t_k \rangle [b]_W.$$

Since $\tilde{W}$ is an extension of W by $\emptyset$ and $[\xi_1]_W = [X_j]_{\tilde{W}}$, the result follows from the inductive hypothesis.

subcase 1.3: a_1 is a variable symbol occurring in b

For some $z \in \mathbf{X} \cup \mathbf{Y}$, $(a_1, z) \in b$. Thus, $a_1[b]_W = [z]_W$ and $\tilde{W} = \mathit{unify\text{-}value}(z, \mathit{advance}(2, W))$. The proof is the same as for subcase 1.1, except that when $[z]_W$ and $[\xi_1]$ are unifiable, τ is replaced by the mgu provided by Lemma 3.14.

subcase 1.4: a_1 is a variable symbol not occurring in b

$\tilde{W} = \mathit{step}(W) = \mathit{unify\text{-}variable}(z, \mathit{advance}(2, W))$, $\tilde{t} = t$, $\tilde{r} = r$, and $\tilde{b} = \{(a_1, z)\} \cup b$, where

$$z = \begin{cases} \mathit{new\text{-}perm}(b), & \text{if } a_1 \in p \\ \mathit{new\text{-}temp}(b, r, n), & \text{if } a_1 \notin p. \end{cases}$$

Since W is compatible with b and p, $[\tilde{b}]_{\tilde{W}} = [b]_W\tau$, where $\tau = \{(a_1, [\xi_1]_W)\}$. Thus, $\tilde{L} = cdr(L)\tau$, and $\tilde{M} = cdr(M)$. The result follows from the observation that if U is an extension of $\tilde{W}$ by $\tilde{\theta}$, where θ is a mgu for $\tilde{L}$ and $\tilde{M}$ that includes $[b' - \tilde{b}]_U$, then (since we may assume that a_1 does not occur in $\tilde{\theta}$) $\theta = \tau\tilde{\theta}$ is a mgu for L and M that includes $[b' - b]_U$.

Case 2: $a = \mathtt{NIL}$

In this case, $W(\mathbf{p})$ points to $\mathit{append}(\mathit{compile\text{-}literal}(\mathtt{GET}-, t, r, b, p, \emptyset, n), c)$ According to the definition of *compile-literal*, there are four subcases. In each of these, $\ell = 0$, $\tilde{t} = cdr(t)$, and $\tilde{r} = cdr(r)$, :

subcase 2.1: t_1 is a constant

Here, $\tilde{a} = \mathtt{NIL}$, $\tilde{b} = b$, and $\tilde{W} = \mathit{get\text{-}constant}(t_1, X_{i_1}, \mathit{advance}(3, W))$. If $[X_{i_1}]_W$ and $t_1[b]_W = t_1$ are not unifiable, then the proposition holds as in subcase 1.1. Otherwise, $\tilde{W}$ is an extension of W by

$$\tau = \begin{cases} \{([X_{i_1}]_W, t_1)\}, & \text{if } [X_{i_1}]_W \text{ is a variable symbol} \\ \emptyset, & \text{if } [X_{i_1}]_W = t_1. \end{cases}$$

Thus, $\tilde{L} = \langle [X_{i_2}]_{\tilde{W}}, \ldots, [X_{i_k}]_{\tilde{W}} \rangle = \mathit{cdr}(L)\tau$ and $\tilde{M} = \mathit{cdr}(M)\tau$. The result folows as in subcase 1.1.

subcase 2.2: t_1 is a compound term

Let $t_1 = \langle f, e_1, \ldots, e_m \rangle$, where f is a function symbol of arity m. Then $\tilde{W} = \mathit{get\text{-}structure}((f, X_{i_1}), \mathit{advance}(3, W))$. We may assume that $dx = \mathit{deref}(W(|X_{i_1}|_W), W)$ is either a variable or a structure with functor f, for otherwise L and M are not unifiable and W fails. In either case, $\tilde{dx} = \mathit{deref}(dx, \tilde{W})$ is a structure of $\tilde{W}$ with arguments $\zeta_i = H_{\tilde{W}(\mathbf{s})+i}$, $i = 1, \ldots, m$. We have $\tilde{b} = b$, $\tilde{a} = \langle e_1, \ldots, e_m \rangle$,

$$\tilde{L} = \langle [\zeta_1]_{\tilde{W}}, \ldots, [\zeta_m]_{\tilde{W}}, [X_{i_2}]_{\tilde{W}}, \ldots, [X_{i_k}]_{\tilde{W}} \rangle,$$

and

$$\tilde{M} = \langle e_1, \ldots, e_m, t_2, \ldots, t_k \rangle [b]_{\tilde{W}}.$$

Suppose dx is a structure H_g of W, so that $dx = \tilde{dx}$. Then

$$\tilde{W} = \mathit{set\text{-}structure\text{-}pointer}(g, W),$$

$$\mathit{funct}(\mathit{car}(L)) = f = \mathit{funct}(\mathit{car}(M)),$$

$$\tilde{L} = \mathit{append}(\mathit{args}(\mathit{car}(L)), \mathit{cdr}(L)),$$

and

$$\tilde{M} = \mathit{append}(\mathit{args}(\mathit{car}(M)), \mathit{cdr}(M)).$$

Thus, a substitution is a mgu for L and M iff it is a mgu for $\tilde{L}$ and $\tilde{M}$.

In the remaining case, dx is an unbound variable of W and $\tilde{W}(dx) = \tilde{dx}$ is a structure of $\tilde{W}$ at heap address $W(\mathbf{h}) + 1$. In fact, $\tilde{W}$ is an extension of W by $\tau = \{(N(dx), [\tilde{dx}]_{\tilde{W}})\}$, where $[\tilde{dx}]_{\tilde{W}} = \langle f, N(\zeta_1), \ldots, N(\zeta_m) \rangle$. If

$$\tilde{L} = \mathit{append}(\langle N(\zeta_1), \ldots, N(\zeta_m) \rangle, \mathit{cdr}(L)\tau)$$

and

$$\tilde{M} = append(\langle e_1, \ldots, e_m\rangle [b]_W \tau, cdr(M))$$

are not unifiable, then W fails, and since none of the variable symbols $N(\zeta_i)$ occurs in L or M, $cdr(L)\tau$ and $cdr(M)\tau$ must not be unifiable. But then neither are L and M, for if $L\theta = M\theta$, then $\tau\theta = \theta$, and hence $cdr(L)\tau\theta = cdr(M)\tau\theta$. Thus, we may assume $W \to U$, where U is an extension of $\tilde{W}$ by $\tilde{\theta}$, and $\tilde{\theta}$ is a mgu for $\tilde{L}$ and $\tilde{M}$. But then U is an extension of W by $\tau\tilde{\theta}$, and hence also by

$$\theta = \{(x, y) \in \tau\tilde{\theta} | x \notin \{N(\zeta_1), \ldots, N(\zeta_m)\}\}.$$

We shall show that θ is a mgu for L and M.

First, since

$$[X_{i_1}]_W \theta = N(dx)\theta = [\tilde{dx}]_{\tilde{\theta}} = t_1[b]_W \tau\tilde{\theta} = t_1[b]_W \theta,$$

$L\theta = M\theta$. Now suppose that $L\sigma = M\sigma$, and let $\tilde{\sigma} = \phi\sigma$, where

$$\phi = \{(N(\zeta_i), e_i[b]_W) | i = 1, \ldots, m\}.$$

Then $\tilde{L}\tilde{\sigma} = \tilde{M}\tilde{\sigma}$, and hence $\tilde{\sigma} = \tilde{\theta}\tilde{\rho}$ for some ρ. Let

$$\rho = (\sigma - \{(N(\zeta_i), N(\zeta_i)\tilde{\rho}) | i = 1, \ldots, m\})\{(N(\zeta_i), N(\zeta_i)\sigma) | i = 1, \ldots, m\}.$$

To show that $\sigma = \theta\rho$, let x be any variable symbol. If $x = \zeta_i$, $1 \le i \le m$, then $x\sigma = x\rho = x\theta\rho$. If $x = N(dx)$, then $x\sigma = t_1[b]_W \sigma = x\tau\phi\sigma = x\tau\tilde{\sigma} = x\tau\tilde{\theta}\rho = x\theta\rho$. In all other cases, $x\sigma = x\tilde{\sigma} = x\tilde{\theta}\rho = x\tau\tilde{\theta}\rho = x\theta\rho$.

subcase 2.3: t_1 occurs in b

For some $z \in \mathbf{X} \cup \mathbf{Y}$, $(t_1, z) \in b$. Thus, $t_1[b]_W = [z]_W$ and $\tilde{W} = get\text{-}value(z, advance(3, W))$. The proof is the same as for subcase 2.1, except that when $[z]_W$ and $[X_{i_1}]$ are unifiable, τ is replaced by the mgu provided by Lemma 3.14.

subcase 2.4: t_1 is a variable symbol not occurring in b

$\tilde{W} = get\text{-}variable(z, advance(, W))$ and $\tilde{a} = \texttt{NIL}$. The proof is the same as for subcase 1.4, with t_1 in place of a_1.□

Similarly, Lemma 4.3 is a special case of the following, with $\ell = 0$, $r = \langle X_1, \ldots, X_k\rangle$, and $b = \emptyset$:

Lemma 4.6 *Let* $a = \langle a_1, \ldots, a_\ell \rangle$ *and* $t = \langle t_1, \ldots, t_k \rangle$ *be lists of terms,* $n \in \mathbf{N}$, $r = \langle X_{i_1}, \ldots, X_{i_k} \rangle$ *a list of temporary symbols,* $p \subset \mathbf{V}$, b *be a symbol table that respects* p *and* n, $b' = \mathit{extend\text{-}table}(t, r, b, p, n)$, *and* $v \subseteq p$ *such that for all* $x \in v$, $(x, z) \in b$ *for some* $z \in \mathbf{Y}$. *Let* W *be a directed WAM state compatible with* b *and* p *such that* $W(\mathbf{p})$ *points to*

$$\mathit{append}(\mathit{compile\text{-}term}(\texttt{PUT-}, a, t, r, b, p, v, n), c)$$

and for $i = 1, \ldots, \ell$, $H_{W(\mathbf{s})+i}$ *is an unbound variable of* W. *Suppose that for each member* X_{i_j} *of* r, *if* $i_j > n$, *then* X_{i_j} *does not occur in* b *and* $W(A_{i_j})$ *is an unbound heap variable* H_{m_j} *of* W, *where the* m_j *are distinct from each other and from* $W(\mathbf{s}) + i$, $i = 1, \ldots, \ell$.

Then $W \rightarrow U$, *where*

a) $U(\mathbf{p})$ *points to* c,

b) U *is an extension of* W *by* $(\phi \cup \psi)\theta$, *where*

$$\phi = \{(N(H_{W(\mathbf{s})+i}), a_i[b']_U) | i = 1, \ldots, \ell\},$$

$$\psi = \{(N(H_{m_j}), t_j[b']_U) | X_{i_j} \text{ is a member of } r \text{ and } i_j > n\},$$

and θ *is a renaming substitution such that* $[b' - b]_U \subseteq \theta$ *and for each* $(x, y) \in \theta$ *there is a variable* ξ *of* U *such that* $N(\xi) = y$ *and if* ξ *is a variable of* W, *then* ξ *is reserved w.r.t.* b,

c) if ξ *is a free variable of* W *w.r.t.* b *and* $\xi \neq |z|_U$ *for all* $(x, z) \in b'$, *then* ξ *is a free variable of* U *w.r.t.* b'; *in particular,* U *is compatible with* b' *and* p,

d) $[X_{i_j}]_U = t_j[b']_U$ *for* $j = 1, \ldots, k$,

e) for each $(x, z) \in b' - b$, *if* x *occurs in a compound term that occurs in* t, *then* $U(|z|_U)$ *is stable.*

f) if $j \leq n$ *and* X_j *does not occur in* r, *or if* $j > n$ *and* X_j *does occur in* r, *then* $W(A_j) = U(A_j)$.

Suppose further that for each $x \in p - v$ *that occurs in* t, *either i) there exists a pair* $(x, z) \in b$ *such that* $W(|z|_W)$ *is stable, or ii) the first occurrence of* x *in* t *is in a compound term. Then for* $j = 1, \ldots, k$, $U(|X_{i_j}|_U)$ *is stable.*

The proof of this lemma is a case analysis that mirrors the proof of Lemma 4.5.

4.4 Compiling Programs and Goals

Let $C = \langle \ell_0, \ldots, \ell_s \rangle$ be a program clause with head ℓ_0 and body $\langle \ell_1, \ldots, \ell_s \rangle$. Let $p = perms(C)$ be a set of order m and let $u = uns(C)$. For $i = 0, \ldots, s$, let q_i be the predicate symbol $\chi(\ell_i)$, let t_i be the argument list $args(\ell_i)$, and let $n_i = length(t_i)$.

Naturally, ℓ_0 is compiled in the context of the trivial symbol table $\emptyset$. The symbol table constructed during this compilation is

$$b_1 = extend\text{-}table(t_0, \emptyset, p, n_1).$$

Note that none of the temporary symbols $X_1, \ldots, X_{n_1}$ occurs in b_1, as the corresponding argument registers are reserved for the arguments of the first subroutine call.

The leading goal literal ℓ_1 (if $s > 0$) is compiled in the context of the symbol table b_1. In general, a goal literal ℓ_i is compiled in the context of some symbol table b_i, which will be required to have the following property: every variable symbol occurring in both $\langle \ell_0, \ldots, \ell_{i-1} \rangle$ and $\langle \ell_i, \ldots, \ell_s \rangle$ also occurs in b_i. For $i > 1$, this holds for the subset of the symbol table constructed during the compilation of ℓ_{i-1} consisting of all pairs corresponding to permanent variable symbols. Thus, for $i > 1$,

$$b_i = \{(x, z) \in extend\text{-}table(t_{i-1}, b_{i-1}, p, n_{i-1}) | z \in \mathbf{Y}\}.$$

The WAM code generated by the clause C is a list returned by the function *compile-clause*, defined in Figure 5. This list takes one of three forms, depending on the number s of goal literals of C:

$s = 0$:

$compile\text{-}head\text{-}literal(t_0, \emptyset, 0)$
⟨`PROCEED`⟩

$s = 1$:

$compile\text{-}head\text{-}literal(t_0, \emptyset, n_1)$
$compile\text{-}goal\text{-}literal(t_1, b_1, \emptyset, \emptyset)$
⟨`EXECUTE`, q_1⟩

$s \geq 2$:

⟨`ALLOCATE`, m⟩
$compile\text{-}head\text{-}literal(t_0, p, n_1)$
$compile\text{-}goal\text{-}literal(t_1, b_1, p, \emptyset)$
⟨`CALL`, q_1⟩
$\vdots$
$compile\text{-}goal\text{-}literal(t_{s-1}, b_{s-1}, p, \emptyset)$
⟨`CALL`, q_{s-1}⟩
$compile\text{-}goal\text{-}literal(t_s, b_s, p, u)$
⟨`DEALLOCATE`⟩
⟨`EXECUTE`, q_s⟩

As described in Lemma 4.2, executing the code for ℓ_0 has the effect of attempting to match the list t_0 with a list of arguments represented by the contents of argument registers. If this attempt fails, backtracking occurs. Otherwise, the code generated by the body of C is executed.

If the body of C is `NIL`, then it generates the single instruction ⟨`PROCEED`⟩. Otherwise, for each goal literal ℓ_i, a call to *compile-goal-literal* returns the code for writing the argument list t_i into argument registers, as described by Lemma 4.3. This is followed by a `CALL` instruction or, in the case of the final goal literal, an `EXECUTE` instruction. If C has more than one goal literal, then there are two additional instructions for allocating and deallocating an environment.

The function *compile-goal*, on which Definition 4.1 depends, is also defined in Figure 5. In this definition, g is a goal clause and R is assumed to be an arbitrary predicate symbol of arity n, where $vars(g) = \langle v_1, \ldots, v_n \rangle$. The value returned is *compile-clause(c)*, where c is the clause with head $(R, v_1, \ldots, v_n)$ and body g. Note that the validity of the definition rests on the observation that *compile-clause(c)* is independent of $\chi(c)$.

The code for the definition of a predicate symbol q is returned by *compile-predicate*, defined in Figure 6. The arguments of this function are a list

```
Function compile-body(g,b,p,u,e);
begin
   t ← args(car(g)); n ← length(t)
   if g = NIL then return ⟨PROCEED⟩
   else if cdr(g) = NIL then
      begin
         c ← ⟨EXECUTE, χ(g)⟩; if e then c ← append(⟨DEALLOCATE⟩, c);
         return append(compile-goal-literal(t,b,p,u),c)
      end
   else
      begin
         c ← append(compile-goal-literal(t, b, p, ∅), ⟨CALL, χ(g)⟩);
         b ← {(x, z) ∈ extend-table(t, b, p, n)|z ∈ Y};
         return append(c,compile-body(cdr(g),b,p,u,e))
      end
end.

Function compile-clause(c);
begin
   t ← args(head(c)); n ← length(args(car(body(c))));
   g ← compile-head-literal(t, perms(c), n);
   if length(body(c)) >= 2
      then begin
               g ← append(⟨ALLOCATE, length(perms(c))⟩, g); e ← TRUE
           end
      else e ← FALSE;
   b ← extend-table(t, ∅, perms(c), n);
   return append(g,compile-body(body(c),b,perms(c),uns(c),e))
end.

Function compile-goal(g);
return compile-clause(((R,vars(g)),g)).
```

Figure 5: Compiling a Clause

$d = \langle C_1, \ldots, C_k \rangle$ of program clauses comprising the definition of q and a natural number i. This second argument represents the address in the code area at which the compiled code for q is to reside. It is required, at least in the case $k > 1$, in order to compute the arguments of the `TRY-ME-ELSE` and `RETRY-ME-ELSE` instructions.

Function *compile-alternatives(l,i)*;
begin
  $c \leftarrow compile\text{-}clause(car(l))$;
  if $cdr(l) =$ `NIL`
    then return $append(\langle$`TRUST-ME-ELSE-FAIL`$\rangle, c)$
    else begin
        $j \leftarrow length(c) + i + 2$;
        $c \leftarrow append(\langle$`RETRY-ME-ELSE`$, j\rangle, c)$;
        return *append(c,compile-alternatives(cdr(l),j))*
      end
end.

Function *compile-predicate(d,i)*;
begin
  if $d =$ `NIL` **then return** $\langle$`BACKTRACK`$\rangle$
  else if $cdr(d) =$ `NIL` **then return** *compile-clause(car(d))*
  else begin
      $c \leftarrow compile\text{-}clause(car(d))$; $j \leftarrow length(c) + i + 2$;
      $c \leftarrow append(\langle$`TRY-ME-ELSE`$, j\rangle, c)$;
      return $append(c, compile\text{-}alternatives(cdr(d), j))$
    end
end.

Figure 6: Compiling a Predicate

In the case $k = 0$, the predicate symbol q has no defining clauses and the compiled code for q consists simply of the instruction $\langle$`BACKTRACK`$\rangle$. If $k = 1$, the code for q is the same as the code for its sole defining clause C_1. The more interesting case $k > 1$ involves the instructions for creating, modifying, and deallocating choice points:

$$
\begin{array}{ll}
 & \langle \texttt{TRY-ME-ELSE}, A_1 \rangle \\
 & \textit{compile-clause}(C_1) \\
A_1 : & \langle \texttt{RETRY-ME-ELSE}, A_2 \rangle \\
 & \textit{compile-clause}(C_2) \\
 & \vdots \\
A_{k-2} : & \langle \texttt{RETRY-ME-ELSE}, A_{k-1} \rangle \\
 & \textit{compile-clause}(C_{k-1}) \\
A_{k-1} : & \langle \texttt{TRUST-ME-ELSE-FAIL} \rangle \\
 & \textit{compile-clause}(C_k)
\end{array}
$$

Note that the argument A_1 of the initial `TRY-ME-ELSE` instruction is the address at which the code following the code for C_1 is to be stored. Similarly, for $j = 2, \ldots, k-1$, the argument A_j of the `RETRY-ME-ELSE` instruction preceding the code for C_j is the address at which the code following the code for C_j is to be stored.

The following is a consequence of Lemmas 3.4 and 3.6:

Lemma 4.7 *Let d be a list of program clauses of length at least 2 and let W be a WAM state with* $W(\mathbf{p}) = k$. *Suppose that either*

a) k points to compile-predicate(d,k) w.r.t. W, or

b) k points to compile-alternatives(d,k) w.r.t. W and $backtrack(W) \approx W$.

Let $W' = step(W)$. *Then* $W'(\mathbf{p})$ *points to compile-clause(car(d)) w.r.t. W and* $backtrack(W') \approx jump(j, W')$, *where j points to*

$$compile\text{-}alternatives(cdr(d), j)$$

w.r.t. W.

We turn now to the function *compile-program* of Figure 7. If $preds(P) = \langle q_1, \ldots, q_k \rangle$, then it follows from the definitions of Figure 7 that since *compile-program(P)* is loaded, so is the list $\langle q_1, a_1, \ldots, q_k, a_k \rangle$, where $a_i \in \mathbf{N}$ points to $compile\text{-}predicate(def(q_i, p), a_i)$ for $i = 1, \ldots, k$. Thus, by Definitions 3.35 and 3.36,

Lemma 4.8 *Let q be a predicate symbol that occurs in P. If for some state W,* $W' = execute(q, W)$, *then* $W'(\mathbf{p})$ *points to*

$$compile\text{-}predicate(def(q, P), W'(\mathbf{p})).$$

Function *compile-defs(q,p,i)*;
begin
  if $q =$ `NIL` **then return** `NIL`
  else begin
      $c \leftarrow \textit{compile-predicate}(def(car(q), p), i)$;
      return $\textit{append}(c, \textit{compile-defs}(cdr(q), p, length(c) + i))$
    end
end.

Function *get-addresses(q,p,i)*;
begin
  if $q =$ `NIL` **then return** `NIL`
  else begin
      $j \leftarrow i + length(\textit{compile-predicate}(def(car(q), p), i)$;
      return $\textit{append}(\langle car(q), i\rangle, \textit{get-addresses}(cdr(q), p, j))$
    end
end.

Function *compile-program(p)*;
begin
  $q \leftarrow preds(p)$; $i \leftarrow 2 \times length(q) + 1$;
  return *append(get-addresses(q,p,i),compile-defs(q,p,i))*
end.

Figure 7: Compiling a Program

4.5 Verification of the Compiler

The proof of Theorem 4.1 is based on a correspondence between interpreter states, as defined in Section 2.3, and certain WAM states that occur during the execution of a program. We begin by characterizing states in which the code for a goal clause is about to be executed, i.e., immediately after executing either a) the code for the head of a program clause with a nontrivial body or b) the `PROCEED` instruction. Such a state is called a *goal state* and in case b), it is also called a *continuation state*. We define these two simultaneously with the following definition:

Definition 4.10 *Let W be a directed WAM state, let H be a goal clause, and let F be a set of variables of W. W is a* **goal state supporting** H **with free variables** F *if either i)* $H =$ `NIL`, $F = \emptyset$, *and W is a success state, or ii)* $H = append(J, K)$, $J \neq$ `NIL`, $F = D \cup E$, *and $W(\mathbf{p})$ points to compile-body(f,b,p,u,e), where for some program clause C of P,*

a) f is a non-`NIL` *tail of body(C);*

b) b is a symbol table that respects p and $arity(\chi(J))$, such that each variable symbol that occurs in both f and $C - f$ also occurs in b, and such that if X_i is a temporary symbol occurring in b, then $W(A_i)$ is an object of W;

c) $p = perms(C)$;

d) $u = uns(C)$;

e) $e =$ `TRUE` *if $length(body(C)) > 1$; otherwise, $e =$* `FALSE`*;*

f) $J = f[b]_W\tau$, where τ is a renaming for C such that no variable symbol that occurs in $C\tau$ occurs in K or P or range(N);

g) V is a continuation state supporting K with free variables E, where

$$V = \begin{cases} proceed(deallocate(W)), & \text{if } e = \texttt{TRUE} \\ proceed(W), & \text{if } e = \texttt{FALSE}; \end{cases}$$

h) if $e =$ `TRUE` *and p is a set of order m, then W has an environment of order m and D is the set of all reserved variables of W w.r.t. b; otherwise, $D = \emptyset$;*

i) if $e =$ `TRUE`*, then for each $(x, z) \in b$ such that x occurs either in head(C) or in a compound term occurring in $body(C) - f$, $W(|z|_W)$ is stable;*

j) F is a set of free variables of W w.r.t. b.

If W satisfies condition i), or condition ii) with $e =$ `TRUE`, then W is also called a **continuation state**.

Lemma 4.9 *Let W be a goal state supporting H with free variables F and let $\xi \in \mathbf{H} \cup \mathbf{S}$. If $N(\xi)$ occurs in H, then ξ is an unbound variable of W and $\xi \notin F$.*

Proof: We may assume that $H = append(J, K)$ and $F = D \cup E$ as in case $ii)$ of Definition 4.11. The proof is by induction on $W(\mathbf{e})$. First note that if $N(\xi)$ occurs in J, then according to part $f)$ of the definition, $N(\xi)$ occurs in $[b]_W$ and the result follows from $b)$ and $j)$. We may assume, therefore, that $N(\xi)$ occurs in K.

Consider the case $e =$ `TRUE`. Here, $V = proceed(deallocate(W))$ is a goal state supporting K with free variables E. By inductive hypothesis, ξ is an unbound variable of V, and hence of W, and $\xi \notin E$. But then ξ cannot be an unstable variable of W, hence $\xi \notin D$ either.

Suppose that $e =$ `FALSE`, so that $D = \emptyset$ and $V = proceed(W)$ is a continuation state supporting K with free variables E. Since $V(\mathbf{e}) = W(\mathbf{e})$, we may revert to the case $e =$ `TRUE` and conclude that ξ is an unbound variable of V, and hence of W, and $\xi \notin E$.□

Lemma 4.10 *Let W be a continuation state supporting H with free variables F, let θ be a substitution such that each variable symbol occurring in θ occurs in either P or range(N), and let W' be an extension of W by θ with $W'(\mathbf{p}) = W(\mathbf{p})$ such that every element of F is a free variable of W' w.r.t. $\emptyset$. Then W' is a continuation state supporting $H\theta$ with free variables F.*

Proof: An inductive proof similar to the proof of Lemma 4.9 may be easily constructed once we establish, in the notation of Definition 4.11, that $J\theta = f[b]_W\tau\theta = f[b]_{W'}\tau$. To this end, let x be a variable symbol occurring in f. If x occurs in b, then $x[b]_W\tau\theta = x[b]_W\theta = x[b]_W\theta\tau = x[b]_{W'}\tau$. If x does not occur in b, then $x[b]_W\tau\theta = x\tau\theta = x\tau = x[b]_{W'}\tau$.□

Next, we characterize the states that exist immediately after $a)$ a `CALL` or `EXECUTE` instruction has been executed or $b)$ backtracking has occurred:

Definition 4.11 *Let W be a directed WAM state and let q be a predicate symbol such that $W(\mathbf{a}) = arity(q) = n$. Let L be a list of program clauses of characteristic q. Let H be a goal clause with car$(H) = \langle q, [X_1]_W, \ldots, [X_n]_W \rangle$ and such that proceed(W) is a continuation state supporting cdr(H).*

a) W is a **calling state supporting** *H if $W(\mathbf{p})$ points to compile-predicate$(L, W(\mathbf{p}))$.*

b) W is a **backtrack state supporting** *H if $W(\mathbf{p})$ points to compile-alternatives$(L, W(\mathbf{p}))$, $L \neq$ `NIL`, and backtrack$(W) = W$.*

Lemma 4.11 *Let W be a calling state that supports a goal clause K such that $W(\mathbf{p})$ points to compile-clause(C) for some program clause C with $\chi(C) = \chi(K)$. If head(C) and car(K) are not unifiable, then W fails. Otherwise, $W \rightarrow W'$, where backtrack$(W') \approx$ backtrack(W) and for some mgu θ for head(C) and car(K), $[\sigma]_{W'}$ is the restriction of $[\sigma]_W\theta$ to G and W' is a goal state supporting the resolvent of $C\rho$ and K via θ, where $C\rho$ is a variant of C with no variables in common with K or $G[\sigma]_W$.*

Proof: Let $\chi(K) = q$, $arity(q) = m$, and $\tilde{r} = \langle [X_1]_W, \ldots, [X_m]_W \rangle$. Then according to Definition 4.11, $car(K) = (q, \tilde{r})$ and for some set E of variables of W, $proceed(W)$ is a continuation state supporting $cdr(K)$ with free variables E. Let $t = args(head(C))$, $n = length(args(car(body(C))))$,

$$b = extend\text{-}table(t, \emptyset, perms(C), n),$$

and

$$g = compile\text{-}head\text{-}literal(t, perms(C), n).$$

If $length(body(C)) \geq 2$, then let $e =$ `TRUE` and

$$h = append(\langle \texttt{ALLOCATE}, length(perms(C)) \rangle, g);$$

otherwise, let $e =$ `FALSE` and $h = g$. According to the definition of *compile-clause*, $W(\mathbf{p})$ points to *append(h,compile-body(body(C),b,perms(C),uns(C),e))*. Let

$$W_1 = \begin{cases} step(W) & \textit{if } \mathrm{e} = \texttt{TRUE} \\ W & \textit{if } \mathrm{e} = \texttt{FALSE}. \end{cases}$$

Then by Lemma 3.5, $backtrack(W_1) \approx backtrack(W)$.

If $head(C)$ and $car(K)$ are not unifiable, then neither are $\tilde{r}$ and t, and the result follows from Lemma 4.2. Therefore, we may assume that head(C) and car(K), and hence $\tilde{r}$ and t, are unifiable. By Lemma 4.2, $W_1 \rightarrow W_2$, where W_2 is an extension of W_1 by some mgu θ for $\tilde{r}$ and t such that $[b]_W \subset \theta$, and $W_2(\mathbf{p})$ points to *compile-body(body(C),b,perms(C),uns(C),e)*. By Definition 3.28, $backtrack(W_2) \approx backtrack(W_1) \approx backtrack(W)$ and $[\sigma]_{W_2}$ is the restriction of $[\sigma]_W\theta$ to G.

By part c) of Lemma 4.2, each variable in E is a free variable of W_2 w.r.t. b. If $e = \texttt{FALSE}$, then $proceed(W_2)$ is an extension of $proceed(W_1) = proceed(W)$ by θ, and hence

$$proceed(W_2)(\mathbf{p}) = W_2(\mathbf{c}) = W(\mathbf{c}) = proceed(W)(\mathbf{p}).$$

We may assume that θ satisfies the hypothesis of Lemma 4.10, allowing the conclusion that $proceed(W_2)$ is a continuation state supporting $cdr(K)\theta$ with free variables E. On the other hand, if $e = \texttt{TRUE}$, then $proceed(deallocate(W_2))$ is an extension of $proceed(deallocate(W_1))$ by θ. By Lemma 3.3,

$$deallocate(W_1) \approx advance(2, W),$$

and hence

$$proceed(deallocate(W_1)) \approx proceed(W).$$

Thus, by the same reasoning as in the case $e = \texttt{FALSE}$, $proceed(deallocate(W_2))$ is a continuation state supporting $cdr(K)\theta$ with free variables E.

Suppose that $body(C) = \texttt{NIL}$. Then $W_2(\mathbf{p})$ points to $\langle\texttt{PROCEED}\rangle$ and $W \rightarrow proceed(W_2)$. Let $W' = proceed(W_2)$. Then W' is a continuation state supporting $cdr(K)\theta$, which is the resolvent of $C\tau$ and K via θ.

We may assume, therefore, that $body(C) \neq \texttt{NIL}$. Let τ be a renaming for C such that no variable occurring in $C\tau$ occurs in $cdr(K)\theta$, P, or $range(N)$. Let

$$\rho = \{(u, v) \in \tau | u \textit{ does not occur in } head(C)\}$$

and let

$$J = body(C)[b]_{W'}\tau = body(C)\rho[b]_{W'} = body(C)\rho\theta.$$

Then J is the resolvent of $C\rho$ and K via θ, and $C\tau$ has no variables in common with either K or $range(N)$. We shall show that according to Definition 4.11, $W' = W_2$ is a goal state supporting

$$H = append(J, cdr(K)\theta).$$

First note that by Lemma 4.1, $b = \{(x_1, z_1), \ldots, (x_s, z_s)\}$ respects $perms(C)$ and $arity(\chi(body(C)))$ and $\{x_1, \ldots, x_s\}$ is the set of all variable symbols that occur in $head(C)$. In the case $e = \texttt{FALSE}$ (i.e., $length(body(C)) = 1$), it follows that W' is a goal state supporting H with free variables E.

Finally, in the case $e = \texttt{TRUE}$, $g) - i)$ of Definition 4.10 must be verified. Let m be the order of the set $perms(C)$. Since W_1 has an environment of order m, so does W_2. Let D be the set of all reserved variables of W' w.r.t. b. By part c) of Lemma 4.2, each element of D is a free variable of W' w.r.t. b. For $1 \leq i \leq m$, $|X_i|_{W_1}$ is a stable object of W_1, and hence by Lemma 4.2, $|z_i|$ is a stable object of W' for $1 \leq i \leq s$. Thus, W' is a goal state supporting H with free variables $D \cup E$.□

Next, we define a correspondence between WAM states of the various types defined above and interpreter states:

Definition 4.12 *Let W be a WAM state and L be a list of program clauses. W* **uses** *L if any of the following holds:*

a) W is a goal state supporting H and $L = def(\chi(H), P)$.

b) W is a calling state and $W(\mathbf{p})$ points to compile-predicate$(L, W(\mathbf{p}))$.

c) W is a backtrack state and $W(\mathbf{p})$ points to compile-alternatives$(L, W(\mathbf{p}))$.

Definition 4.13 *Let $R = \langle\langle G_k, \ldots, G_1\rangle, \langle L_k, \ldots, L_1\rangle, \langle \sigma_k, \ldots, \sigma_1\rangle\rangle$ be an interpreter state and let W be a goal state, a calling state, or a backtrack state. W* **realizes** *R if*

a) W supports G_k;

b) W uses L_k;

c) $\sigma_k = [\sigma]_W$;

d) W' has a choice point iff $k > 1$, where

$$W' = \begin{cases} W, & \text{if } W \text{ is a goal state or a calling state} \\ \textit{trust-me-else-fail}(W), & \text{if } W \text{ is a backtrack state;} \end{cases}$$

if $k > 1$, then backtrack(W') is a backtrack state that realizes

$$\langle\langle G_{k-1}, \ldots, G_1\rangle, \langle L_{k-1}, \ldots, L_1\rangle, \langle \sigma_{k-1}, \ldots, \sigma_1\rangle\rangle.$$

The following is an obvious consequence of the definitions:

Lemma 4.12 *Let W be a WAM state that realizes an interpreter state R. If $W' \approx W$, then W' realizes R.*

Lemma 4.13 *Let W be a calling state or a backtrack state that realizes an interpreter state R. If $succ(\bar{R})$ is defined, then $W \to W'$, where W' realizes R' and $succ(\bar{R}) = \bar{R}'$. If $succ(\bar{R})$ is undefined, then W fails.*

Proof: Let $R = \langle\langle G_k, \ldots, G_1\rangle, \langle L_k, \ldots, L_1\rangle, \langle \sigma_k, \ldots, \sigma_1\rangle\rangle$, where L_k is a list of program clauses of some common characteristic q. We proceed by induction, assuming the statement true if k is replaced by $k-1$ or if k is held constant and $L_k \neq \texttt{NIL}$ is replaced by $cdr(L_k)$.

Let $m = arity(q)$. Then $W(\mathbf{a}) = m$, $car(G_k) = \langle q, [X_1]_W, \ldots, [X_m]_W\rangle$, $proceed(W)$ is a goal state supporting $cdr(G_k)$, and $\sigma_k = [\sigma]_W$. Let

$$R' = \langle\langle G_{k-1}, \ldots, G_1\rangle, \langle L_{k-1}, \ldots, L_1\rangle, \langle \sigma_{k-1}, \ldots, \sigma_1\rangle\rangle.$$

Case 1: W is a calling state.

$W(\mathbf{p})$ points to $compile\text{-}predicate(L_k, W(\mathbf{p}))$, W has a choice point iff $k > 1$, and if so, then $backtrack(W)$ is a backtrack state that realizes R'.

Suppose $L_k = \texttt{NIL}$. Then $W(\mathbf{p}))$ points to $\langle\texttt{BACKTRACK}\rangle$ and $W \to W' = step(W) = backtrack(W)$ if $k > 1$. If so, then $succ(\bar{R}) = succ(\bar{R}')$ and the result follows by induction. If not, then $k = 1$ and $succ(\bar{R})$ is undefined.

Let $C = car(L_k)$. If $L_k = \langle C\rangle$, then $W(\mathbf{p})$ points to $compile\text{-}clause(C)$ and the result follows by induction from Lemma 4.11. Thus, we may assume $cdr(L_k) \neq \texttt{NIL}$. Let $W_1 = step(W)$ and $W_2 = backtrack(W_1)$. By Lemma 4.7, $W_1(\mathbf{p})$ points to *compile-clause(C)* and $W_2 \approx jump(j, W_1)$, where j points to $compile\text{-}alternatives(cdr(L_k), j)$. Since $W_2(\mathbf{c}) = W(\mathbf{c})$, $proceed(W_2)$ is a continuation state supporting $cdr(G_k)$, and hence W_2 is a backtrack state supporting G_k using $cdr(L_k)$.

Let $W_3 = trust\text{-}me\text{-}else\text{-}fail(W_2)$. Since

$$W_2 \approx jump(j, W_1) = try\text{-}me\text{-}else(j, jump(j, W)),$$

$W_3 \approx jump(j, W)$ by Lemma 3.7. It follows that W_3 has a choice point iff $k > 1$, and if so, then $backtrack(W_3$ is a backtrack state that realizes R'. Since $\sigma_k = [\sigma]_W = [\sigma]_{W_2}$, W_2 realizes

$$\langle\langle G_k, \ldots, G_1\rangle, \langle cdr(L_k), L_{k-1}, \ldots, L_1\rangle, \langle \sigma_k, \ldots, \sigma_1\rangle\rangle.$$

The result follows by induction from Lemma 4.11 and Definition 2.7.

Case 2: W is a backtrack state.

$W(\mathbf{p})$ points to *compile-alternatives*$(L_k, W(\mathbf{p}))$, $L_k \neq \texttt{NIL}$, and

$$backtrack(W) = W.$$

Let $W_1 = step(W)$, $W_2 = backtrack(W_1)$, and $W_3 = trust\text{-}me\text{-}else\text{-}fail(W)$. If $k > 1$, then $backtrack(W)$ realizes

$$\langle\langle G_{k-1}, \ldots, G_1\rangle, \langle L_{k-1}, \ldots, L_1\rangle, \langle \sigma_{k-1}, \ldots, \sigma_1\rangle\rangle.$$

If $k = 1$, then $backtrack(W_3)$ is undefined.

If $cdr(L_k) \neq \texttt{NIL}$, then by Lemma 4.7, $backtrack(W_1)$ is a backtrack state that realizes

$$\langle\langle G_k, \ldots, G_1\rangle, \langle cdr(L_k), L_{k-1}, \ldots, L_1\rangle, \langle \sigma_k, \ldots, \sigma_1\rangle\rangle.$$

If $cdr(L_k) = \texttt{NIL}$, then $W_2 = advance(1, W_3)$. In either case, the result follows from Lemma 4.11.□

Lemma 4.14 *Let W be a goal state that realizes an interpreter state R. If W is not a success state, then $W \rightarrow W'$, where W' is a calling state that realizes R' and $\bar{R}' = \bar{R}$.*

Proof: Let $R = \langle\langle G_k, \ldots, G_1\rangle, \langle L_k, \ldots, L_1\rangle, \langle \sigma_k, \ldots, \sigma_1\rangle\rangle$ and $q = \chi(G_k)$. Then W is a goal state supporting G_k using $L_k = def(q, P)$ and $\sigma_k = [\sigma]_W$. If $k = 1$, then W does not have a choice point; otherwise, $backtrack(W)$ realizes $\langle\langle G_{k-1}, \ldots, G_1\rangle, \langle L_{k-1}, \ldots, L_1\rangle, \langle \sigma_{k-1}, \ldots, \sigma_1\rangle\rangle$.

Thus, $G_k = append(J, K)$ and $W(\mathbf{p})$ points to *compile-body*(f, b, p, u, e), where $J, K, f, b, p, u, e, C, \tau, F, D$, and E are as described in Definition 4.11. Let $t = args(car(f))$, $n = arity(q)$, $b' = extend\text{-}table(t, b, p, n)$, and

$$b'' = \{(x, z) \in b' | z \in \mathbf{Y}\}.$$

Then $W(\mathbf{p})$ points to

$$append(compile\text{-}goal\text{-}literal(t, b, p, v), c),$$

where

$$v = \begin{cases} u, & \text{if } cdr(f) = \texttt{NIL} \\ \emptyset, & \text{if not} \end{cases}$$

and

$$c = \begin{cases} \langle \texttt{EXECUTE}, q\rangle, \text{ if } e = \texttt{FALSE} \\ \langle \texttt{DEALLOCATE}, \texttt{EXECUTE}, q\rangle, \text{ if } e = \texttt{TRUE} \text{ and } cdr(f) = \texttt{NIL} \\ append(\langle \texttt{CALL}, q\rangle, compile\text{-}body(cdr(f), b'', p, u, e)), \text{ if } cdr(f) \neq \texttt{NIL}. \end{cases}$$

In any case, Lemma 4.3 applies with $k = n$. Let U and θ be as described in Lemma 4.3. We shall show that $U \to W'$, where W' is a calling state supporting $G_k\hat{\tau}\theta\tau$ using L_k.

First, note that we may assume that any variable occurring in θ must also occur in either $range(N)$ or b'. One consequence of this is that Lemma 4.10 applies. Another is that if x is a variable symbol that occurs in f, then $x[b]_W\theta = x[b']_{W'}$. To prove this, note that if x occurs in b, then $x[b]_W\theta = x[b]_{W'} = x[b']_{W'}$, while if x occurs in f but not in b, then $x[b]_W\theta = x\theta = x[b'-b]_{W'} = x[b']_{W'}$.

Case 1: $e = \texttt{FALSE}$

In this case, $cdr(f) = \texttt{NIL}$. Let

$$W' = step(U) = execute(q, advance(U, 2)).$$

Then $proceed(W')$ is an extension of $proceed(W)$ by θ, and by Lemma 4.10, $proceed(W')$ is a continuation state supporting $K\theta = K\hat{\tau}\theta\tau$. $W'(\mathbf{p})$ points to $compile\text{-}predicate(L_k, W'(\mathbf{p}))$ and $W'(\mathbf{a}) = n$. By Lemma 4.3, $[X_j]_{W'} = t_j[b']_{W'}$ for $j = 1, \ldots, n$, and hence

$$\begin{aligned} \langle q, [X_1]_{W'}, \ldots, [X_n]_{W'}\rangle &= car(f)[b']_{W'} = car(f)[b]_W\theta \\ &= car(f)[b]_W\tau\hat{\tau}\theta\tau = car(J)\hat{\tau}\theta\tau. \end{aligned}$$

Thus, W' is a calling state supporting $append(J, K)\hat{\tau}\theta\tau = G_k\hat{\tau}\theta\tau$.

Case 2: $e = \texttt{TRUE}$, $cdr(f) = \texttt{NIL}$

Let $W' = step(step(U)) = execute(q, advance(deallocate(U), 2))$. Then $proceed(W')$ is an extension of $proceed(deallocate(W))$ by θ, and by Lemma 4.10, $proceed(W')$ is a continuation state supporting $K\theta = K\hat{\tau}\theta\tau$. The conclusion will follow as in Case 1, once it is verified that for $j = 1, \ldots, n$, $U(A_j)$ is a stable object of U, so that $[X_j]_{W'}$ is defined.

Let $x \in p - u$ such that x occurs in f and the first occurrence of x in f is not in a compound term. Since x is safe w.r.t. C, x occurs either in $head(C)$

or in a compound term occurring in $body(C) - f$, and hence $(x, z) \in b$ for some z. According to Lemma 4.3, it suffices to show that $W(|z|_W)$ is a stable object of W. But this follows from Definition 4.10, part i).

Case 3: $cdr(f) \neq \texttt{NIL}$

In this case, $e = \texttt{TRUE}$. Let $W' = step(U) = call(q, advance(U, 2))$. As in Case 1,

$$\langle q, [X_1]_{W'}, \ldots, [X_n]_{W'} \rangle = car(J)\hat{\tau}\theta\tau = car(G_k\hat{\tau}\theta\tau),$$

and we must show that $W_1 = proceed(W')$ is a continuation state supporting

$$cdr(G_k\hat{\tau}\theta\tau) = append(cdr(J)\hat{\tau}\theta\tau, K\hat{\tau}\theta\tau).$$

Since

$$W_1(\mathbf{p}) = W'(\mathbf{c}) = advance(U, 2)(\mathbf{p}),$$

$W_1(\mathbf{p})$ points to $compile\text{-}body(cdr(f), b'', p, u, e)$. Let D' be the set of all reserved variables of W_1 w.r.t. b'' and let $F' = D' \cup E$. We refer to Definition 4.10, substituting W', $cdr(f)$, b'', $cdr(J)\hat{\tau}\theta\tau$, $K\hat{\tau}\theta\tau$, F', and D' for W, f, b, J, K, F, and D, respectively:

a), c), d), and e) hold trivially.

b) This follows from Lemma 4.1.

f) $cdr(J)\hat{\tau}\theta\tau = cdr(f)[b]_W\tau\hat{\tau}\theta\tau = cdr(f)[b]_W\theta\tau = cdr(f)[b']_{W'}\tau = cdr(f)[b'']_{W'}\tau$

g) $proceed(deallocate(W_1))$ is an extension of $proceed(deallocate(W))$ by θ and

$$\begin{aligned} proceed(deallocate(W_1))(\mathbf{p}) &= deallocate(W_1)(\mathbf{c}) \\ &= deallocate(W)(\mathbf{c}) \\ &= proceed(deallocate(W))(\mathbf{p}). \end{aligned}$$

By Lemma 4.10 and Lemma 4.3, part c), $proceed(deallocate(W_1))$ is a continuation state supporting $K\theta = K\hat{\tau}\theta\tau$ with free variables E.

h) Since W has an environment of order m, so does W_1.

i) If x is a variable symbol occurring in both $cdr(f)$ and $C - cdr(f)$, then either x occurs in b or x is a permanent variable symbol occurring in $car(f)$. In either case, x occurs in b''. Let $(x, z) \in b''$ such that x occurs in either $head(C)$ or a compound term occurring in $body(C) - cdr(f)$. If x occurs in a compound term occurring in $car(f)$, then $W_1(|z|_{W_1})$ is stable by Lemma 4.3, part f). Otherwise, $W(|z|_W) = W(|z|_{W_1})$ is already stable, hence so is $W_1(|z|_{W_1})$.

j) This follows from Lemma 4.3, part c).

Thus, in all cases, W' is a calling state supporting $G'_k = G_k\hat{\tau}\theta\tau$ using L_k. Let σ'_k be the restriction of $[\sigma]_{W'}$ to G. Note that

$$G\sigma'_k = G[\sigma]_{W'} = G[\sigma]_W\theta = G[\sigma]_W\hat{\tau}\theta\tau = G\sigma_k\hat{\tau}\theta\tau.$$

Since $backtrack(W') \approx backtrack(W)$, W' realizes

$$R' = \langle\langle G'_k, G_{k-1}, \ldots, G_1\rangle, \langle L_k, \ldots, L_1\rangle, \langle\sigma'_k, \sigma_{k-1}, \ldots, \sigma_1\rangle\rangle.$$

Using Lemma 4.9, we have

$$G'_k\hat{\tau}\hat{\theta}\tau = G_k\hat{\tau}\theta\tau\hat{\tau}\hat{\theta}\tau = G_k\hat{\tau}\theta\hat{\theta}\tau = G_k\hat{\tau}\tau = G_k$$

and similarly

$$G\sigma'_k\hat{\tau}\hat{\theta}\tau = G\sigma_k\hat{\tau}\theta\tau\hat{\tau}\hat{\theta}\tau = G\sigma_k.$$

Thus, $\bar{R}' = \bar{R}$.□

The following is a consequence of Lemmas 4.13 and 4.14:

Lemma 4.15 *Let W be a WAM state that realizes an interpreter state R and let R' be a refutation state. There exists a success state W' such that $W \rightarrow W'$ and W' realizes some representative of $\bar{R}'$ iff there exists a list of interpreter states $\langle R_k, \ldots, R_0\rangle$ such that $R = R_0$, $R' = R_k$, $\bar{R}_i = succ(\bar{R}_{i-1})$ for $1 \leq i \leq k$, and R_i is not a refutation state for $1 < i < k$.*

Finally, we may prove Theorem 4.1. Note that the state W_0 of Definition 4.1 is a calling state that supports the goal clause

$$\langle\langle R, N(H_1), \ldots, N(H_n)\rangle\rangle$$

and such that $W_0(\mathbf{p})$ points to *compile-clause*(C) where

$$C = (\langle R, v_1, \ldots, v_n\rangle, G)$$

and $vars(G) = \langle v_1, \ldots, v_n\rangle$. Also, W_0 does not have a choice point and $[\sigma]_{W_0}$ is a renaming for G. It follows from Lemma 4.11 that $W_0 \rightarrow W'$, where W' is a goal state that realizes some interpreter state equivalent to the state $\langle\langle G\rangle, \langle def(\chi(G), P)\rangle, \emptyset\rangle$. The theorem now follows from Lemma 4.15 by induction on the length of Σ.□

References

[1] Aït-Kaci, H., The WAM: A (Real) Tutorial, Digital Paris Research Laboratory, 1990.

[2] Apt, K. R. and van Emden, M. H., Contributions to the Theory of Logic Programming, *J. ACM.* 29:841-863 (1982).

[3] Boyer, R. S. and Moore, J S., *A Computational Logic Handbook*, Academic Press, New York, 1988.

[4] Bridgeland, D. M., Russinoff, D. M., and Schelter, W. F., A LISP-based Extended Warren Abstract Machine for Expert Reasoning, forthcoming technical report, MCC, Austin, Texas.

[5] Butler, R., Lusk, E. L., Olson, R., and Overbeek, R. A., ANLWAM: A Parallel Implementation of the Warren Abstract Machine, Argonne National Laboratory, Argonne, Illinois, 1989.

[6] Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*, Springer-Verlag, Berlin, 1981.

[7] Gabriel, J., Lindholm, T., Lusk, E. L., and Overbeek, R. A., A Tutorial on the Warren Abstract Machine, ANL-84-84, Argonne National Laboratory, Argonne, Illinois, 1985.

[8] Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984.

[9] Robinson, J. A., *Logic: Form and Function*, Elsevier North Holland, Inc., New York, 1979.

[10] Warren, D. H. D., Applied Logic — its Use and Implementation as a Programming Tool, SRI Technical Note No. 290, SRI International, 1983.

[11] Warren, D. H. D., An Abstract Prolog Instruction Set, SRI Technical Note No. 309, SRI International, 1983.