Journal of Universal Computer Science, vol. 3, no. 4 (1997), 341-376
submitted: 20/12/96, accepted: 21/4/97, appeared: 28/4/97 © Springer Pub. Co.

A Structured Presentation of a Closure-Based Compilation Method for a Scoping Notion in Logic Programming

Keehang Kwon
(DongA University, Korea
kwon@se.donga.ac.kr)

Abstract: We present a systematic reconstruction of a compilation method for an extension to logic programming that permits procedure definitions to be given a scope. At a logical level, this possibility is realized by permitting implications to be embedded in goals. Program clauses that appear in the antecedents of such implications may contain variables that are bound by external quantifiers, leading to non-local variables in procedure declarations. In compiling programs in this extended language, there is, therefore, a need to consider the addition to given programs of program clauses that are parameterized by bindings for some of their variables. A proposed approach to dealing with this aspect uses a *closure* representation for clauses. This representation separates an instance of a clause with parameterized bindings into a skeleton part that is fixed at compile-time and an environment that maintains the part that is dynamically determined. A development of this implementation scheme is provided here by starting with an abstract interpreter for the language and then refining this to arrive at an interpreter that uses the closure representation for clauses. The abstract state machine formalism of Gurevich is used in specifying the interpreters that are of interest at the two different stages. We also justify this refinement by showing that the essential notion of a computation is preserved by the refinement and thus the refinement is a correct one.
Key Words: logic programming, embedded implication, scoping, closure, compilation

1 Introduction

Logic programming has traditionally lacked mechanisms that permit procedure declarations and names to be given a scope. This deficiency is an outcome of using a weak logic as the basis for languages in this paradigm. Without scoping constructs, logic programming does not provide the capability of modularizing code: in particular, a program is typically monolithic, unstructured collection of procedure definitions. This is an unsatisfactory situation from the perspective of developing large programs and, consequently, attention has been given to methods for restricting the extent of particular procedure declarations. One approach that is of interest to us in this paper involves the enrichment of the underlying logic to allow for hypothetical or implication goals [Miller 89, Miller et al. 91]. An implication goal has the form of $D \supset G$ where D is a conjunction of program clauses and G is a goal and is intended to be interpreted in the following fashion: it is to be solved by adding D to the current program and then attempting to solve G. Thus, the availability of the procedure declaration contained in D is to be restricted to the context of trying to solve G.

The scoping ability provided by implication goals is sufficient to support the idea of local definitions as well as a more general notion of modules. An illustration of the former facet is provided by the following definition of the reverse relation between lists:

$$
\begin{array}{l}
(rev(L1, L2) \text{ :- } \\
\qquad (((rev\_aux([], L2)) \wedge \\
\qquad (\forall X \forall L1 \forall L2 (rev\_aux([X|L1], L2) \text{ :- } rev\_aux(L1, [X|L2])))) \\
\qquad\qquad \supset rev\_aux(L1, []))).
\end{array}
$$

The body of this clause contains an implication goal. The clauses that occur in the antecedent of this goal contain an explicit universal quantification over some of their variables. Such a quantification is necessary to distinguish these variables from other "non-local" variables such as $L2$ in the first clause that are intended to be (implicitly) quantified at the outer level. Now, based on the semantics of implication that we have discussed informally above, we see that there is a definition of the predicate rev_aux that is not available at the top-level but that becomes available when solving the goal $rev\_aux(L1, [])$ in the body of rev. As a particular example, solving the query $rev([1, 2, 3], L)$ would result in the clauses

$$
\begin{array}{l}
rev\_aux([], L). \\
\forall X \forall L1 \forall L2 (rev\_aux([X|L1], L2) \text{ :- } rev\_aux(L1, [X|L2])).
\end{array}
$$

being added to the program, subsequent to which an attempt would be made to solve the goal $rev\_aux([1, 2, 3], [])$. Notice that the first of these clauses contains a variable that is, in fact, "tied" to a variable in the query. The given goal will succeed after a sequence of backchaining steps by instantiating this variable to $[3, 2, 1]$ and this value will eventually be returned as the desired result at the top level.

As seen from the example above, implication goals can be used to give a scope to program clauses. However, the use of this device also raises new implementation problems. One problem arises from the fact that the programs to be used in solving different subgoals might be distinct. An efficient method has, therefore, to be found for managing changing program contexts. Special consideration has to be given, in this regard, to the requirement that a program that was operative earlier in the computation must be resurrected upon backtracking. Another problem relates to the issue of compiling clauses that appear in the antecedent of implication goals. Thus, given a goal expression of the form $D(\bar{x}) \supset G$, a method that supports the compilation of the clauses constituting $D(\bar{x})$ needs to be described. As indicated schematically, these clauses may contain variables that are bound in the textual context external to the specific implication goal. A satisfactory compilation method must permit a decoupling of the occurrences of such variables from the quantifiers binding them in the generation of code. However, the code that is produced must include some means for the run-time coordination of different occurrences of variables bound by the same quantifier.

An approach to handling these and related problems has been described in [Nadathur et al. 95]. The proposed scheme utilizes a stack-based approach to managing dynamically changing program contexts and has built into it relevant book-keeping devices for backtracking. Further, it treats the problem of tied variables in program clauses by using a notion similar to that of *closures* used in implementations of functional programming languages; the code generated for a program clause may then contain a segment of code for initializing bindings for some variables from an associated environment. These ideas are given specific substance by describing an abstract machine for the extended logic programming language and by outlining a compilation method in its context.

The work in this paper is part of a larger effort to provide a rational reconstruction of, and to thereby verify, these implementation ideas [Kwon 94]. The approach that we have used is to start with a high-level description of an interpreter for the language of interest and to refine this in well-motivated and understandable ways so as to arrive eventually at a description that is based on executing low-level instructions on a corresponding abstract machine. We have used the abstract state machine formalism of Gurevich [Gurevich 91] as a specification language in this task. This formalism is especially suited to our task since it permits perspicuous yet mathematically precise description of machines to be provided at various levels of granularity. There is, furthermore, a well-developed verification methodology that can be used in conjunction with this formalism: despite differences in detail, we can use standard techniques to show that successive refinements to a given machine preserve a relevant essential notion a computation.

In this paper we provide an analysis in the described form of only one component of the implementation scheme for the overall language. In particular, we consider only the transformation of embedded program clauses that underlies their separate compilation and their treatment as closures. This abbreviated discussion is motivated partly by reasons of space. However, we also believe that the component of the implementation approach that we treat here is of independent interest. The notion of embedded program clauses and the possibility of quantifiers of differing scope are present in a variety of proposed extensions to logic programming [Gabbay and Reyle 84, Giordano et al. 88, Hodas and Miller 94, McCarty 88a, McCarty 88b, Miller 94, Monteiro and Porto 89]. The ideas that we present here will, we believe, be an important component of the proper treatment of such languages.

The rest of this paper is organized as follows. Section 2 summarizes the important notions pertaining to abstract state machines that are used in this paper. Section 3 presents the extended logic programming language that is of interest and describes a nondeterministic interpreter for it. Section 4 presents an abstract state machine specification of a deterministic version of this interpreter. In Section 5, we describe a refinement of the deterministic interpreter which introduces the notion of environments. This refinement is essential to the reuse of code. We then present a preprocessing of clauses and goals that is central to the scheme for compiling embedded clauses. These discussions provide the basis for the presentation of an abstract state machine that incorporates the idea of environments. In Section 6, we verify that this machine is equivalent to the earlier deterministic interpreter. We conclude the paper in Section 7.

2 The Abstract State Machine Formalism

To understand the basic notion that underlies abstract state machines, let us consider the execution of an algorithm. This execution can be represented, without loss of generality, as a sequence of states $S_0, S_1, \ldots, S_n, \ldots$ with S_0 being the initial state. The abstract state machine approach views each state S_i as a many-sorted mathematical structure, *i.e.*, a number of finite disjoint sets called *universes* and functions (including constants) on Cartesian products of universes. Each state is represented by the states of these universes, together with the values of the associated functions. There are several advantages in adopting this view. First, it is suitable for utilizing the simplicity found in most computation processes. As long as the components of the structure are properly set up, a set of simple transition rules usually suffices to describe the transition from S_i to S_{i+1}. In fact, only three elementary transitions are currently permitted: a universe extension, a universe contraction, and a function update. A universe extension amounts to adding a new element to the universe, while a universe contraction is the opposite operation—it removes a specified element from the universe. A function update amounts to updating the value of the function at the given inputs. Second, the abstraction levels of the components (*i.e.*, universes and functions) can be tailored to any desired level of granularity. As a particular example, we can specify the semantics of a language at many different abstraction levels including one that corresponds to a high-level presentation of the language and another that corresponds to its implementation. When these different levels of description are related to each other by a process of refinement, they can be used in a verification of the language implementation.

Because of these attributes of abstract state machines, they have been successfully used in describing the operational semantics of various programming languages including Modula-2 [Gurevich and Morris 88], Smalltalk [Blakley 92], Occam [Gurevich and Moss 90], Prolog [Börger 90a, Börger 90b] as well as in proving the correctness of a Prolog machine [Börger and Rosenzweig 94].

2.1 The Basic Terminology

We define formally some of the notions relating to abstract state machines.

Definition 2.1. A many-sorted first-order algebra $\mathcal{A}$ consists of a number of disjoint sets called *universes* and functions on the Cartesian product of these sets. The collection of these function symbols is referred to as the *signature* of $\mathcal{A}$. A function of arity zero is called a *distinguished element* or *constant*.

Following Gurevich [Gurevich 91], we assume that both the universes and the associated signature are fixed throughout the computation. Among the configurations, some are identified, as usual, as *initial* and *final* states.

The following notation corresponding to universes will be useful:

Definition 2.2. For any universe U, we shall use the notation U^* to stand for the universe of finite lists of elements of U. An empty list is denoted by nil and elements of U^* will be generated by means of an infix constructor ::. For instance, $a_1 :: a_2 :: a_3 :: nil$ represents a list of three elements a_1, a_2 and a_3 of some universe. We shall often use $[a_1, \ldots, a_n]$ as a shorthand notation for $a_1 :: \ldots :: a_n :: nil$ and [] for an empty list. An element in the Cartesian product of universes $U_1 \times \ldots \times U_n$ is written as $\langle a_1, a_2, \ldots, a_n \rangle$ provided that each a_i is in the universe U_i for $1 \leq i \leq n$.

Definition 2.3. Relative to an algebra $\mathcal{A}$, a *transition rule* of is an expression of the form

```
if condition
then  update1; update2; ...; updaten
endif
```

where *condition* is a Boolean expression, and each $update_i$, $1 \leq i \leq n$, is one of the following three kinds:

(i) A *function update* of the form $f(t_1, \ldots t_n) := t$ where f be a function symbol of $\mathcal{A}$ and $t_1, \ldots, t_n, t$ are terms relative to the signature of $\mathcal{A}$. An update of this form corresponds to the redefinition of the function f at the given arguments.

(ii) A *universe contraction* of the form `dispose(`tmp`)`, where tmp is an element of some universe of $\mathcal{A}$. When this operation is executed, tmp is deleted from the universe to which it belongs and functions defined on this element are made undefined.

(iii) A *universe extension* of the form

$$\texttt{extend}\ U\ \texttt{by}\ tmp_1, \ldots, tmp_k\ \texttt{with}$$
$$F_1;\ F_2;\ \ldots;\ F_n$$
$$\texttt{endextend}$$

which first extends the universe U by adding new elements $tmp_1, \ldots, tmp_k$ to this universe, and then *simultaneously* performs the function updates $F_1, \ldots, F_n$ which may depend on tmp_i's, $1 \leq i \leq k$.

A transition rule of this form is to be interpreted as follows: if the *condition* is true, then each update will be executed *simultaneously*, producing a new algebra (with the same signature).

It is convenient to allow the syntax of transition rules to be extended to include constructs such as `if-then-else`, `let-in-endlet`, `case-endcase`, with the usual meanings. These forms are reducible to the original format and so can be dispensed with in a more elaborate presentation.

Definition 2.4. An *abstract state machine* is a pair $\langle \mathcal{A}, \mathcal{T} \rangle$ consisting of a finite, many-sorted, first-order algebra $\mathcal{A}$ which serves as configurations and a finite set of transition rules $\mathcal{T}$ with respect to which the configuration evolves in discrete time.

2.2 Showing the Equivalence of Abstract State Machines

In later sections we shall be interested in observing that two abstract state machines M and M' that differ significantly in their detailed structure are, nevertheless, equivalent in a relevant computational sense. We describe this situation by saying that M and M' simulate each other, where the notion of simulation is defined as follows:

Definition 2.5. An abstract state machine M with an initial state A_0 *simulates* another abstract state machine M' with an initial state B_0 if the following hold:

(1) If B_0 reaches a final success state through a sequence of transitions in M', then A_0 reaches a final success state through a sequence of transitions in M. Furthermore, it is the case that they produce identical observable outputs.

(2) If B_0 reaches a final failure state through a sequence of transitions in M', then A_0 reaches a final failure state through a sequence of transitions in M.

To establish that M simulates M', we shall typically proceed as follows: First, we shall identify a subset of the states reachable from an initial state in the two machines as their *essential* states and we define a *state relation* Φ which maps essential states of M' to essential states of M. Then we show that the following properties hold of Φ and the essential states:

(a) Φ maps an initial state of M' to an initial state of M, every final success state of M' to a final success state of M with identical observable outputs, and every final failure state of M' to a final failure state of M,
(b) the set of essential states for M and M' are properly chosen so that any transition sequence from an initial state to a final state in the two abstract state machines can be decomposed into a sequence of transitions between essential states, and
(c) If B_i is an essential state of M' and A_j is an essential state of M that B_i is mapped to by Φ, then, for any essential state B_{i+1} of M' that can be reached by a single transition (under decomposition) from B_i, there is an essential state A_k of M that B_{i+1} is mapped to by Φ and that can be reached by a finite sequence of transitions from A_j in M.

The requirement stated in (c) is shown pictorially below, where S is a single transition (under decomposition) in M' and T is a corresponding transition sequence in M:

$$\begin{tikzcd}
A_j \arrow[r, "T"] & A_k \\
B_i \arrow[u, "\Phi"] \arrow[r, "S"'] & B_{i+1} \arrow[u, "\Phi"']
\end{tikzcd}$$

If we can carry out these steps successfully, we can actually conclude that M simulates M'. A detailed inductive argument can be provided in support of this conclusion with the commutativity of the above figure being used in an obvious manner in the inductive step.

3 A Language with Embedded Implications

Two classes of formulas are central to our description of the logic programming language that is of interest in this paper. These are G- and D-formulas given by the following syntax rules:

$$G ::= A \mid G \wedge G \mid G \vee G \mid D \supset G$$
$$C ::= A \mid G \supset A \mid \forall x C$$
$$D ::= C \mid C \wedge D.$$

In the rules above, A represents an atomic formula. The C-formulas defined here are a subclass of first-order hereditary Harrop formulas [Miller et al. 87]. In the programming language to be considered, G-formulas will function as queries and collections of C-formulas will constitute programs. For this reason, we refer to a G-formula as a goal or query, to a C-formula as a clause, to a D-formula as a program clause, and to a collection of clauses as a program.[1] As explained in [Miller et al. 91], the programming language based on Horn clause logic can also be characterized in a fashion similar to that done here, with the main difference that implications are not permitted in G-formulas. This symbol is included in the language we consider here so as to provide a notion of scope.

In the framework of [Miller et al. 91], the notion of programming consists of describing relationships between objects through a program and of querying such a specification through a G-formula. We will show exactly how this is done by presenting an operational semantics for this language, *i.e.*, the rules for solving a query in the context of a given program. These rules correspond to each possible case for the top-level logical symbol in the query and have the effect of producing a new query and a new program. Thus, the operational semantics induces a notion of computational state given by a program and a query. To represent such a state, we employ structures of the form $\mathcal{P} \longrightarrow G$ where $\mathcal{P}$ is a listing of closed clauses and G is a G-formula. We refer to these structures as *sequents*, and the idea of solving a query from a set of closed clauses corresponds to that of constructing a derivation for an appropriate sequent.

Some terminology and notation are needed before we can define the notion of a derivation precisely. We shall write $\mathcal{F}(F)$ to denote the set of free variables in a formula F. We also have to consider the notion of a substitution for some of the free variables in an expression.

Definition 3.1. A *substitution* θ is a mapping from a finite set of variables to the set of terms and is written as $\{\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$. The *domain* of such a substitution, denoted by $dom(\theta)$, is the set $\{x_1, \ldots, x_n\}$. The *restriction* of θ to $\mathcal{V}$ is the substitution θ' such that $\theta' = \{\langle x, t\rangle \mid \langle x, t\rangle \in \theta \text{ and } x \in \mathcal{V}\}$. A substitution $\theta = \{\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$ is a (variable) *renaming* substitution if t_i is a distinct variable for $1 \leq i \leq n$.

Given a substitution θ, we write $\theta(F)$ to denote the application of θ to a formula F. Such an application must be done carefully to avoid the usual capture problems. We write $\theta_1 \circ \theta_2$ to denote the the composition of θ_1 and θ_2, *i.e.*, $\theta_1 \circ \theta_2(x) = \theta_1(\theta_2(x))$. We also note that the composition of substitutions is an associative operation, *i.e.*, $(\theta_1 \circ \theta_2) \circ \theta_3 = \theta_1 \circ (\theta_2 \circ \theta_3)$. We shall often use the notation $[t/x]F$ to denote the application of the singleton substitution $\{\langle x, t\rangle\}$ to a formula F. Finally, a formula F_1 is said to be an alphabetic variant of another

[1] Although no explicit syntax is provided for this purpose, existential quantification may also be present in goals. Thus, a clause of the form $\forall x(G(x) \supset A)$, is equivalent (in intuitionistic logic) to $(\exists x G(x) \supset A)$, provided that x does not appear free in H.

formula F_2 if F_1 is obtained by replacing some (possibly no) subparts of the form $\forall y G$ or $\exists y G$ of F_2 by $\forall z([z/y]G)$ or $\exists z([z/y]G)$, where z is a variable which is not free in G.

In describing the idea of a derivation, we shall need to define the notion of an instance of a clause.

Definition 3.2. An *instance* of a clause of the form $\forall x_1 \ldots \forall x_n A$ or $\forall x_1 \ldots \forall x_n$ $(G \supset A)$ is any formula that can be written as $\theta(A)$ or $\theta(G \supset A)$, where θ is a substitution $\{\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$ for some choices of terms $t_1, \ldots, t_n$.

The rules for solving queries in our language are "goal-directed" in the sense that the next rule to be used depends on the top-level logical symbol of the goal formula.

Definition 3.3. Let G be a query and let $\mathcal{P}$ be a finite set of clauses. Then a derivation is constructed for $\mathcal{P} \longrightarrow G$ by using one of the following rules:

SUCCESS	By noting that G is identical to an instance of a clause in $\mathcal{P}$.
BACKCHAIN	By picking an instance of a clause in $\mathcal{P}$ of the form $G_1 \supset G$ and constructing a derivation for $\mathcal{P} \longrightarrow G_1$.
AND	If G is $G_1 \wedge G_2$, by constructing derivations for $\mathcal{P} \longrightarrow G_1$ and $\mathcal{P} \longrightarrow G_2$.
OR	If G is $G_1 \vee G_2$, by constructing a derivation for either $\mathcal{P} \longrightarrow G_1$ or $\mathcal{P} \longrightarrow G_2$.
AUGMENT	If G is $(C_1 \wedge \ldots \wedge C_n) \supset G_1$, by constructing a derivation for $C_1, \ldots, C_n, \mathcal{P} \longrightarrow G_1$.

The above rules essentially allow the connectives in goal formulas to be interpreted as search primitives. Thus, $\vee$ and $\wedge$ can be used to specify OR and AND branches in a search. The symbols $\supset$, on the other hand, provide a scoping mechanism: it allows for the augmentation of the program in the course of solving a query. The above notion of "computation-as-search" is referred to as *uniform provability* in [Miller et al. 91] and is justified there as a basis for programming in logic.

A standard way of obtaining an answer (or an output) from a computation in logic programming is by solving a goal with free variables. Thus, a goal with free variables may be interpreted as a request to produce the instantiations for the free variables that lead to a successful solution. Our ultimate interest is in a procedure for carrying out computations of the kind described above and for extracting results from these computations. The rules for constructing derivations provide a structure for such a procedure, but additional mechanisms are needed. One problem arises from solving a query with free variables, where a proper instantiation that yields a solution must be picked. A standard technique for dealing with this is to delay the instantiations of such variables until information is available for making an appropriate choice. This effect is usually achieved by replacing the free variables by placeholders whose values are determined at a later stage through the process of unification. Thus, a goal such as $G(x)$ may be transformed into one of the form $G(X)$ where X is a new "logic" variable that may be instantiated at a later stage. In attempting to solve an atomic goal A, we look for a program clause $\forall y_1 \ldots y_n(G' \supset A')$ such that A unifies with the atomic formula that results from A' by replacing the universally quantified variables by new logic variables. If such a clause is found, the next task becomes that of solving the resulting instance of G'.

We now describe a nondeterministic interpreter for our language. This interpreter refines the operational semantics by incorporating the unification procedure into the process of solving a goal from a program. We describe this abstract interpreter by means of a transition system. The states of this transition system are given by a tuple of the form $\langle \mathcal{G}, \mathcal{V}, \theta \rangle$. In the tuple, $\mathcal{G}$ represents a set of tuples $\langle G, \mathcal{P} \rangle$ where G is a goal formula and $\mathcal{P}$ is a finite set of clauses. $\mathcal{V}$ represents the set of free variables appearing in the state and θ denotes a substitution. Transition rules in the system of interest are those given by the following definition.

Definition 3.4. Given a state $\langle \mathcal{G}_1, \mathcal{V}_1, \theta_1 \rangle$, the state $\langle \mathcal{G}_2, \mathcal{V}_2, \theta_2 \rangle$ can be obtained from it in one of the following ways:

(1) Suppose that $\mathcal{G}_1$ is $\{\langle A, \mathcal{P} \rangle\} \cup \mathcal{G}'$ and that C is a clause $\forall x_1 \ldots \forall x_n A'$ in $\mathcal{P}$. Let $\rho = \{\langle x_1, w_1 \rangle, \ldots, \langle x_n, w_n \rangle\}$ be a renaming substitution such that, for $1 \leq i \leq n$, w_i is a distinct variable not in $\mathcal{V}_1$. If $\theta_1(A)$ and $\theta_1 \circ \rho(A')$ are unifiable with a most general unifier σ, then the new state may be obtained by setting $\mathcal{G}_2$ to $\mathcal{G}'$, $\mathcal{V}_2$ to $\mathcal{V}_1 \cup \{w_1, \ldots, w_n\}$ and θ_2 to $\sigma \circ \theta_1$.

(2) Suppose that $\mathcal{G}_1$ is $\{\langle A, \mathcal{P} \rangle\} \cup \mathcal{G}'$, and that C is a clause $\forall x_1 \ldots \forall x_n (G \supset A')$ in $\mathcal{P}$. Let $\rho = \{\langle x_1, w_1 \rangle, \ldots, \langle x_n, w_n \rangle\}$ be a renaming substitution such that, for $1 \leq i \leq n$, w_i is a distinct variable not in $\mathcal{V}_1$. If $\theta_1(A)$ and $\theta_1 \circ \rho(A')$ are unifiable with a most general unifier σ, then the new state may be obtained by setting $\mathcal{G}_2$ to $\{\langle \rho(G), \mathcal{P} \rangle\} \cup \mathcal{G}'$, $\mathcal{V}_2$ to $\mathcal{V}_1 \cup \{w_1, \ldots, w_n\}$ and θ_2 to $\sigma \circ \theta_1$.

(3) If $\mathcal{G}_1$ is $\{\langle G_1 \wedge G_2, \mathcal{P} \rangle\} \cup \mathcal{G}'$, then the new state may be obtained by setting $\mathcal{G}_2$ to $\{\langle G_1, \mathcal{P} \rangle, \langle G_2, \mathcal{P} \rangle\} \cup \mathcal{G}'$, $\mathcal{V}_2$ to $\mathcal{V}_1$ and θ_2 to θ_1.

(4) If $\mathcal{G}_1$ is $\{\langle G_1 \vee G_2, \mathcal{P} \rangle\} \cup \mathcal{G}'$, then the new state may be obtained by setting $\mathcal{G}_2$ to either $\{\langle G_1, \mathcal{P} \rangle\} \cup \mathcal{G}'$ or $\{\langle G_2, \mathcal{P} \rangle\} \cup \mathcal{G}'$, $\mathcal{V}_2$ to $\mathcal{V}_1$, and θ_2 to θ_1.

(5) If $\mathcal{G}_1$ is $\{\langle D \supset G, \mathcal{P} \rangle\} \cup \mathcal{G}'$ and D is of the form $C_1 \wedge \ldots \wedge C_n$, then the new state may be obtained by setting $\mathcal{G}_2$ to $\{\langle G, \mathcal{P} \cup \{C_1, \ldots, C_n\} \rangle\} \cup \mathcal{G}'$, $\mathcal{V}_2$ to $\mathcal{V}_1$, and θ_2 to θ_1.

To complete the description of our transition system, we need to specify its initial and final states. The initial state is obviously dependent on the given query and program. Let G be a goal and $\mathcal{P}$ be a finite set of closed clauses. Then an initial state relative to G and $\mathcal{P}$ is given by $\langle \{\langle G, \mathcal{P} \rangle\}, \mathcal{V}, \emptyset \rangle$ where $\mathcal{V}$ is the set of free variables appearing in G. A final state in the transition system is any state $\langle \mathcal{G}', \mathcal{V}', \theta' \rangle$ where $\mathcal{G}'$ is an empty set. A successful computation can then be explained as a sequence of transitions that starts at an initial state and reaches a final state.

Definition 3.5. Let G be a goal, let $\mathcal{P}$ be a set of closed clauses and let $\langle \mathcal{G}_1, \mathcal{V}_1, \theta_1 \rangle$ be an initial state corresponding to G and $\mathcal{P}$. A *derivation* relative to G and $\mathcal{P}$ is then a sequence $\langle \mathcal{G}_i, \mathcal{V}_i, \theta_i \rangle_{1 \leq i \leq n}$ where, for $1 \leq i < n$, $\langle \mathcal{G}_{i+1}, \mathcal{V}_{i+1}, \theta_{i+1} \rangle$ is obtained from $\langle \mathcal{G}_i, \mathcal{V}_i, \theta_i \rangle$ by virtue of the transition rules in Definition 3.4. Such a derivation is called a *derivation of* G from $\mathcal{P}$ if, in addition, $\langle \mathcal{G}_n, \mathcal{V}_n, \theta_n \rangle$ is a final state, *i.e.*, if $\mathcal{G}_n = \emptyset$. In this case, θ_n restricted to $\mathcal{F}(G)$ is referred to as the corresponding *answer substitution*.

The following proposition, whose proof is immediate from the discussions in [Nadathur 93][2], relates our transition system to an existing deduction system.

Proposition 1. *Let $\mathcal{P}$ be a set of closed clauses and let G be a query whose free variables are included in $\{x_1, \ldots, x_n\}$. Then the following holds:*

(i) *Suppose there is a derivation of G from $\mathcal{P}$ with answer substitution θ. Then there is a proof in intuitionistic logic for $\mathcal{P} \longrightarrow \theta(G)$.*
(ii) *Suppose there is a proof in intuitionistic logic for $\mathcal{P} \longrightarrow \sigma(G)$, for some substitution σ. Then there is a derivation of G from $\mathcal{P}$ with an answer substitution θ that is more general than σ. Moreover, such a derivation can be obtained by picking the next tuple to be processed in an arbitrary fashion.*

Up to this point, we have described a transition system in which nondeterminism is present in several places. To be specific, there is a choice concerning which disjunct of a disjunctive goal is to be solved in the OR rule, and a choice concerning which program clause is to be selected in solving an atomic goal. Furthermore, there is also a choice concerning which tuple is to be picked next. An abstract interpreter must, in principle, explore every possible alternative whenever nondeterminism is present. However, by virtue of Proposition 1, we see that nondeterminism in picking up the next tuple to be processed can be replaced, without a loss of completeness, by some fixed rule.

From now on, we assume that the interpreter always picks the leftmost tuple where $\mathcal{G}$ is represented as a sequence of tuples to be solved. The other two nondeterminisms are similar to those in the case of Horn clause logic and can be handled, as usual, by a depth-first search with the following rules: disjunctive goals will be considered in left-to-right order and program clauses will be used in the order of presentation. While it is well-known that this has a drawback of incompleteness, it can be very efficiently implemented.

In the next section, we present a specification of a deterministic interpreter based on the choices above.

4 A Specification of a Deterministic Interpreter

Based on the discussions in Section 3, we now present an abstract state machine specification of a deterministic version of the abstract interpreter. The execution strategy of our interpreter is a depth-first search of a tree with backtracking and thus calls for a stack for maintaining possible alternative derivation paths to be explored. We represent this stack using a universe NODE, two distinguished constants *root* and *cnode* for representing the bottom and top element of the stack, and a function $b : \text{NODE} - \{root\} \rightarrow \text{NODE}$ which yields the previous element in the stack for any given node. Thus the stack algebra is of the form

$$(\text{NODE}; root, cnode; b)$$

[2] Our transition system is a simplified variant of the one described in [Nadathur 93]: it dispenses with the elaboration operation, the INSTANCE and the GENERIC rules. In addition, it applies the computed substitution to formulas in a *lazy* way, and it maintains in each tuple a composite rather than an incremental substitution.

where the universe NODE extends dynamically as the computation proceeds.

Our computation model can be seen as a linear layout of the standard depth-first search tree; it creates children of a node on "demand" rather than creating them all in advance. This requires our interpreter to work in two different major modes: *call* mode and *try* mode. In *call* mode, the interpreter tries to solve the goal sequence (*i.e.*, the $\mathcal{G}$ component) associated with a node, while in *try* mode, it tries to initiate an alternative computation thread.

We now briefly describe how our interpreter works in these modes. When, at a given node n, the first tuple of the $\mathcal{G}$ component is called for execution in *call* mode, all possible candidates, if there are more than one, are selected and attached to n as a list $cands(n)$. At this stage, no children of the node will be explicitly created. The mode is then switched to try. In this mode, the interpreter creates a child of n using the first element in $cands(n)$. The computation then proceeds in *call* mode. If control ever returns to n due to backtracking, the computation will proceed in try mode, trying to select the next element in the $cands(n)$ list. If there is one, then it will initiate an alternative computation thread using the element. If there is none, the node n will be abandoned and control will return to its parent node, the node continuing to be set to try; This overall action is usually referred to as *backtracking*. The computation then proceeds in try mode. In addition to these two modes, our interpreter may also proceed in the modes *enter* and $unify$ in the course of solving an atomic goal. The purpose of the *enter* mode is to create a new instance of a selected clause. The purpose of the $unify$ mode is to attempt to unify the head of the selected clause with a given atomic goal. Several functions such as g, i, together with a constant *cll* are newly introduced to support this mode.

We define the "interpreter" machine called $\mathcal{M}_1$ by first specifying its universes and signature and then describing its transition rules. The resulting specification, as we shall see, turns out to be simple and mathematically rigorous.

4.1 Universes and Functions of the Machine $\mathcal{M}_1$

We list below the various universes and functions of the machine $\mathcal{M}_1$ that is intended to correspond to a deterministic version of the interpreter. Recall that the notation U^* where U is a universe represents the universe of sequences of elements of U. This notation is used in the descriptions that follow.

(1) There is a universe $\mathcal{N}$ of natural numbers with the usual arithmetic functions $+,-$. There is also a universe BOOL given by the set $\{true, false\}$.

(2) There are a universe $\mathcal{FV}$ which consists of free variables. Associated with these universes, there is an injective function $w : \mathcal{N} \rightarrow \mathcal{FV}$ which returns an ith free variable given an index i. In addition, there is a universe $\mathcal{BV}$ which consists of bound variables.

(3) There is a universe of predicate symbols. We use PSYMBOL to denote this universe. Further, there are universes such as TERM, GOAL, CLAUSE, and ATOM which represent, respectively, the set of terms, the set of G-formulas, the set of clauses, and the set of atomic formulas.

(4) To handle unification in our machine, we introduce a universe SUBST consisting of substitutions. Related to these universes is a function $unify$, a composition function $\circ$ and a function *apply*. The function $unify$ has

the type $(\text{TERM} \times \text{TERM})^* \rightarrow (\text{SUBST} \cup \{fail\})$. Given a list of disagreement pairs T, the function $unify$ either produces a most general unifier for T, or reports a failure if T is not unifiable. The function $\circ$ produces the composition of two given substitutions and has the type $\text{SUBST} \times \text{SUBST} \rightarrow \text{SUBST}$. Finally, the function *apply* has the type $U \times \text{SUBST} \rightarrow U$ where U may be either ATOM, CLAUSE or GOAL. This function is thus parameterized by a choice of domain for U. Given a substitution θ and a suitably determined element e, the function *apply* produces the result obtained by applying θ to the element. We shall often use the notation $\theta(e)$ instead of $apply(e, \theta)$.

(5) Solving an atomic goal requires clauses whose heads are unifiable with the goal to be selected. We assume a function $procdef : \text{CLAUSE}^* \times \text{ATOM} \rightarrow \text{CLAUSE}^*$ that, given a list of clauses $\mathcal{P}$ and an atomic goal A, yields a list of clauses of the form $\forall x_1 \ldots \forall x_n (G \supset A')$ or $\forall x_1 \ldots \forall x_n A'$ in $\mathcal{P}$ such that the predicate symbol of A' and its arity are identical to those of A. We expect that the order in which such clauses are listed respects the order in which they appear in $\mathcal{P}$.

(6) There is a distinguished constant $stop : \{0, 1, -1\}$ which indicates running of the computation, termination with success, or termination due to no more choice points.

(7) There is a distinguished constant $mode : \{try, call, enter, unify\}$ which indicates the mode of the computation, as explained in the informal discussion.

(8) As explained above, we need to represent a "stack". For this, we assume a universe called NODE. We also have two distinguished elements *root* and *cnode* ranging over NODE and a function $b : \text{NODE} \rightarrow \text{NODE}$. Elements of NODE, called *nodes*, will also "contain" information relevant to computation. This is modelled via the following functions called *decorating* functions:

- $\mathcal{G} : \text{NODE} \rightarrow (\text{GOAL} \times \text{CLAUSE}^*)^*$ that yields a list of tuples of the form $\langle G, \mathcal{P} \rangle$ where G is a goal formula and $\mathcal{P}$ is a list of clauses. We refer to each such tuple as a decorated goal and to a list of such tuples as a decorated goal sequence.
- $vi : \text{NODE} \rightarrow \mathcal{N}$ that yields a variable index Vi for any given node. This index is intended to be such that $w(Vi)$ is the first variable that has not been used at a relevant point in the computation.
- $\theta : \text{NODE} \rightarrow \text{SUBST}$ that yields the computed substitution at a relevant point in the computation.
- $cands : \text{NODE} \rightarrow \text{CLAUSE}^*$ that yields a list of candidate clauses to be explored.
- $candg : \text{NODE} \rightarrow \text{GOAL}^*$ that yields a list of candidate goals to be explored.
- $g : \text{NODE} \rightarrow \text{ATOM}$ which is used as a device for storing an atomic goal that is to be solved.
- $i : \text{NODE} \rightarrow \text{CLAUSE}^*$ which is used as a device for storing the program active at a relevant point in the computation.
- $tmode : \text{NODE} \rightarrow \{trycl, trygl\}$ which indicates a submode of computation in *try* mode; the value of this function for a given element of NODE indicates whether the "choice point" is one resulting out of a

disjunctive goal or a multitude of clauses.

It is useful to note that each element of NODE can be thought of as a record, the fields of the record indicating the values of $\mathcal{G}, vi, \theta, cands, candg, g, i, tmode$ and b for that node. It is under this viewpoint that NODE will correspond to a linked stack in the course of a computation, with *cnode* "pointing" to the top of the stack and *root* pointing to the bottom.

(9) There is a constant *cll* : CLAUSE which temporarily stores the selected candidate clause at a relevant point in the computation. This constant can be thought of as a "register".

4.2 The Transition Rules

We complete the description of our abstract state machine by defining the transition rules. Prior to doing this, we define the notions of initial and final states in our algebra. It can be easily seen that these notions together with the transition rules constitute a deterministic rendition of the interpreter defined in Section 3. We also observe that the machine $\mathcal{M}_1$ needs to be parameterized by the choice of program and goal and that a precise identification of the machine being considered requires these to be specified. However, such a specific identification may not be necessary in all situations and we shall use the expression "the machine $\mathcal{M}_1$" in such cases.

Definition 4.1. Let $\mathcal{P}$ be a program and G be a goal such that the set of free variables appearing in G is of the form $\{w(0), w(1), \ldots, w(n-1)\}$ for some positive number n. An *initial state* of the machine $\mathcal{M}_1$ with program $\mathcal{P}$ and goal G is then the state in which

(i) *mode* is set to *call* and *stop* is set to 0, and
(ii) there are two nodes in NODE and *root* and *cnode* "reference" these. The "contents" of these nodes are as follows:
  (a) No functions are defined on the node that *root* corresponds to, *i.e.*, this is a *nil* node.
  (b) $\mathcal{G}(cnode)$ is set to $[\langle G, \mathcal{P}\rangle]$, $vi(cnode)$ is set to n, $\theta(cnode)$ is an empty substitution, and $b(cnode) = root$.

Definition 4.2. A *final success state* of the algebra $\mathcal{M}_1$ with program $\mathcal{P}$ and goal G is a state in which $stop = 1$. As we shall see below, this represents the (first) successful execution of the query. In this case, $\theta(cnode)$ restricted to the free variables of G is referred to as the *answer substitution*. A *final failure state* is a state in which $stop = -1$.

We present transition rules of the interpreter in Figures 1–3. We assume that all transition rules are implicitly guarded under $stop = 0$, *i.e.*, they are applicable only to states in which $stop = 0$. A transition rule t that is guarded by some condition *cond* is actually a rule of the form `if` *cond* `then` t `endif`.

In presenting these rules, we find the following abbreviations useful:

(1) when the argument to a decorating function is *cnode*, we shall suppress it; *e.g.*, we write $\mathcal{G}$ for $\mathcal{G}(cnode)$ and θ for $\theta(cnode)$.
(2) we shall use *backtrack* as an abbreviation for the following transition rule:

(1) the AND rule

```
if mode = call & 𝒢 = ⟨G1 ∧ G2, 𝒫⟩ :: 𝒢';
then  𝒢 := ⟨G1, 𝒫⟩ :: ⟨G2, 𝒫⟩ :: 𝒢'
endif
```

(2) the AUGMENT rule

```
if mode = call & 𝒢 = ⟨(C1 ∧ ... ∧ Cn) ⊃ G, 𝒫⟩ :: 𝒢'
then  𝒢 := ⟨G, C1 :: ... :: Cn :: 𝒫⟩ :: 𝒢'
endif
```

(3) the OR rule

```
if mode = call & 𝒢 = ⟨G1 ∨ G2, 𝒫⟩ :: 𝒢'
then  candg := [G1, G2]; 𝒢 := 𝒢'; i := 𝒫;
      tmode := trygl; mode = try
endif
```

(4) the TRY GOAL rule

```
if mode = try & tmode = trygl
then  case candg of
      []:        backtrack
      G :: Goals:   candg := Goals;
                    extend NODE by t with
                    𝒢(t) := ⟨G, i⟩ :: 𝒢; vi(t) := vi; θ(t) := θ;
                    b(t) := cnode; cnode := t
                    endextend
                    mode = call
      endcase
endif
```

Figure 1: Transition rules for solving complex goals

```
if b(cnode) = root
then  stop := −1
else  cnode := b; mode = try
endif.
```

(3) the symbol & will represent the "logical-and" operation.

The rules presented in the figures are self explanatory, being a straightforward rendition of the deterministic interpreter discussed in Section 3; note that the various transition rules apply only to mutually exclusive states and thus the machine $\mathcal{M}_1$ does in fact correspond to a deterministic interpreter. In understanding the rules, it is perhaps relevant to note the following:

(1) Sets of tuples of the form $\langle G, \mathcal{P}\rangle$ are now maintained as lists; the usual arguments justify such a representation in a "machine" implementation. As a particular consequence, repetition of the same element is allowed in both programs and goal sets.

(2) The mechanism for using clauses has been broken up into three transition rules: SELECTION, ENTER and UNIFY.

(1) the SELECTION rule

`if` $mode = call$ & $\mathcal{G} = \langle A, \mathcal{P}\rangle :: \mathcal{G}'$
`then` $cands := procdef(\mathcal{P}, A)$; $\mathcal{G} := \mathcal{G}'$; $g := A$; $i := \mathcal{P}$;
$tmode := trycl$; $mode = try$
`endif`

(2) the TRY CLAUSE rule

`if` $mode = try$ & $tmode = trycl$
`then` `case` $cands$ `of`
[]: $backtrack$
$C :: Cands$: $cands := Cands$;
`extend` NODE `by` t `with`
$\mathcal{G}(t) := \mathcal{G}$; $vi(t) := vi$; $\theta(t) := \theta$;
$g(t) := g$; $i(t) := i$; $b(t) := cnode$; $cnode := t$
`endextend`;
$cll := C$; $mode = enter$
`endcase`
`endif`

(3) the ENTER rule

`if` $mode = enter$
`then` `let` $cll = \forall x_1 \ldots \forall x_s C'$ `and` $\rho = [\langle x_1, w(vi)\rangle, \ldots, \langle x_s, w(vi+s-1)\rangle]$
`in` $vi := vi + s$; $cll := \rho(C')$ `endlet`;
$mode = unify$
`endif`

(4) the UNIFY rule

`if` $mode = unify$
`then` `case` cll `of`
$G \supset A$: `case` $unify([\langle\theta(g), \theta(A)\rangle])$ `of`
$fail$: $backtrack$
σ : $\theta := \sigma \circ \theta$; $\mathcal{G} := \langle G, i\rangle :: \mathcal{G}$; $mode = call$
`endcase`
A: `case` $unify([\langle\theta(g), \theta(A)\rangle])$ `of`
$fail$: $backtrack$
σ : $\theta := \sigma \circ \theta$; $mode = call$
`endcase`
`endcase`
`endif`

Figure 2: Transition rules related to solving atomic goals

(1) the QUERY SUCCESS rule

`if` $mode = call$ & $\mathcal{G} = []$
`then` $stop := 1$
`endif`

Figure 3: The success rule

Despite the division into SELECTION, ENTER and UNIFY mentioned in (2) above, it will be convenient to combine transitions caused by these three transition rules in analyzing transition sequences. This motivates the following definition.

Definition 4.3. The essential states of $\mathcal{M}_1$ are categorized as follows:

(1) The final success or failure states.
(2) *stop* $= 0$ and *mode* is set in that state to either *try* or *call*.

Theorem 2. *The initial and final states of $\mathcal{M}_1$ are essential states. Furthermore, any transition sequence from an initial state to a final state can be decomposed into a sequence of transitions between essential states.*

Proof. This theorem follows from the definition of essential states of $\mathcal{M}_1$ and a straightforward inspection of the transition rules.

This concludes the specification of a deterministic interpreter for the extended language that is based on a depth-first search of a tree with backtracking. As mentioned in Section 3, the interpreter described here is not *complete* in that it may follow an infinite path even when a successful derivation exists. We note that the specification in this section is modelled closely on the specification of Prolog in [Börger and Rosenzweig 94].

In the next section, we will have to consider clauses and goals that are identical to given ones except for the fact that quantifier prefixes may be reordered. Such clauses and goals are referred to as π-variants of each other, as made clear by the following definition.

Definition 4.4. The notion of a clause being a π-variant of another clause is given as follows:

(1) A clause $\forall x_1 \ldots \forall x_s C$ is a π-variant of the clause $\forall x_{i_1} \ldots \forall x_{i_s} C$ if $i_1, \ldots, i_s$ is a permutation of the sequence $1, \ldots, s$.
(2) $\forall x_1 \ldots \forall x_s C_1$ is a π-variant of any alphabetic variant of $\forall x_{i_1} \ldots \forall x_{i_s} C_2$ where
  (a) $i_1, \ldots, i_s$ is a permutation of the sequence $1, \ldots, s$, and
  (b) C_2 is obtained from C_1 by replacing some (possibly no) subformulas that are clauses by their π-variants.

We use the notation $C \equiv_\pi C'$ to denote that two clauses C and C' are π-variants of each other.

Similarly, the notion of a goal being a π-variant of another goal is given as follows: a goal formula G_1 is a π-variant of any alphabetic variant of another goal G_2 if G_2 is obtained from G_1 by replacing some (possibly no) subformulas that are clauses by their π-variants. We use the notation $G \equiv_\pi G'$ to denote that G and G' are π-variants of each other.

The following lemma, whose proof is straightforward, shows that the machine with program $\mathcal{P}$ and goal G is equivalent to one in which $\mathcal{P}$ and G are replaced by ones that are their π-variants.

Lemma 3. *Let G be a goal and let $\mathcal{P}$ be a program. Further, let G' and $\mathcal{P}'$ be a goal and a program that are π-variants of G and $\mathcal{P}$. Then the machine $\mathcal{M}_1$ with program $\mathcal{P}$ and goal G is equivalent (in the sense of Section 2.2) to the machine $\mathcal{M}_1$ with program $\mathcal{P}'$ and goal G'. In particular, the answer substitutions in the two cases are identical up to variable renaming.*

5 An Environment-based Model

Once a clause is selected in $\mathcal{M}_1$, a new version of it is produced by performing renaming substitutions on it. Performing these substitutions in an eager fashion preempts a sharing of clause and goal structure. This has undesirable consequences from the perspective of space usage and must also be avoided if compilation of clauses is to be possible.

We consider a refinement of $\mathcal{M}_1$ that permits a delaying of substitutions and, consequently, provides a basis for sharing of structure. The standard technique for delaying renaming substitutions is to record them in a data structure called an environment and to read a relevant formula skeleton in conjunction with this environment. This technique is used, for instance, in the WAM [Warren 83]. We adapt this method to our context in this section and also extend it to handle the new feature of our language. The new requirement is to deal with clauses that have free variables in them that are "tied" to other occurrences of these variables; such a clause would arise, for instance, in the course of solving the goal $(D(x) \supset G(x))$.

The actual scheme that is used is reasonably complex and we therefore begin by describing it informally in the first section. The implementation of this scheme requires a preprocessing of clauses and goals that we take up in Section 5.2. We then describe an abstract state machine that incorporates the idea of environments.

5.1 Introduction

One of our objectives is to delay the renaming substitutions that have to be performed on a selected clause in the context of $\mathcal{M}_1$. To understand what is involved in achieving this, consider a clause that is given schematically as

$$\forall z_1 \forall z_2 (G_1 \supset p(z_1)).$$

When this clause is selected, a renaming substitution such as $\{\langle z_1, w(\kappa)\rangle\}, \langle z_2, w(\kappa{+}1)\rangle\}$ would have to be applied on $(G_1 \supset p(z_1))$, assuming κ is the value of vi at the node under consideration. Notice, however, that this substitution can be implicitly performed by "remembering" κ and thinking of z_1 as an "offset" of 0 from κ and of z_2 as an offset of 1 from κ. Of course, this interpretation of z_1 and z_2 must be consistent throughout the clause. Furthermore, the variables $w(\kappa)$ and $w(\kappa{+}1)$ may themselves have to be bound in the course of execution and hence must have some physical space allocated to them. The first requirement is achieved by a preprocessing phase that replaces the quantified variables by a new kind of variables that we call "offset" variables and the second requirement is met (in an actual implementation) by the allocation of an environment.

The major complication to the environment scheme arises from nested clauses and variable dependencies across these. As an illustration of the simplest case, consider a clause of the form

$$\forall z_1((C_1(z_1) \supset G) \supset p(z_1)).$$

Suppose this clause is selected and unification with its head succeeds. This would result in an attempt to solve a suitable instance of the goal $C_1(z_1) \supset G$ which, in turn, results in solving an instance of G after adding an instance $C_1(z_1)$ to the program. Under the scheme being considered, the binding for the variable z_1 would have to be resolved as an offset relative to a base value for the instance of the overall clause. This in general leads to a treatment of a clause in the program as a *closure* consisting of a "skeleton" clause and a base value with respect to which the free variables in the skeleton are determined.

Working with the example being considered a bit further reveals a final issue to be handled. Thus, based on the preceding discussion, we have in our program a clause of the form $\langle C_1(z_1), \kappa_1\rangle$. Assume that this clause is now selected. The first step in using it is that of allocating its environment. This leads to our having to deal with two different base values, one of these being κ_1 and the other being the one for the newly created environment. Moreover, it must be possible to determine the base value with respect to which a given offset variable has to be resolved. This situation gets worse if $C_1(z_1)$ itself contains embedded clauses, *e.g.*, if $C_1(z_1)$ were of the form

$$\forall z_2((C_2(z_1, z_2) \supset G') \supset p'(z_1, z_2)).$$

Our solution to this problem is to reduce the number of base values to 1. This is achieved by resolving *all* variables in a clause as offsets from the same base value and to include an initialization phase, when a clause is selected, that dynamically reconciles the free variables in the clause with their values determined from the surrounding environment. Thus, consider the case of $C_1(z_1)$ in the example just considered. At the level of the clause within which $C_1(z)$ is embedded, the variable z_1 is conceptually substituted for by a new variable of the form $w(\kappa_1 + (i-1))$. Now, suppose z_1 is treated as a variable with offset j within $C_1(z_1)$. Its value within an instance created for $C_1(z_1)$ would then be determined by the value of the variable $w(\kappa_2 + (j-1))$ where κ_2 is the base value for this instance. The initialization phase must therefore result in binding $w(\kappa_2 + (j-1))$ to $w(\kappa_1 + (i-1))$, *i.e.*, in registering a substitution of the form $\{\langle w(\kappa_2 + (j-1), w(\kappa_1 + (i-1))\rangle\}$.

In realizing this final aspect of our scheme, the preprocessing phase must generate a table of initializing bindings corresponding to a clause that is to be used in the dynamic process. Under this final view, a preprocessed clause consists of a tuple $\langle N, IT, Skel\rangle$ where N is the size of the environment, IT is an initialization table and $Skel$ is a (preprocessed) skeleton of the clause. This structure will get linked with a base value during execution to determine an actual clause. We describe the preprocessing phase that produces such a representation for the clauses in the next section and then go on to using it in the execution model.

5.2 Preprocessing clauses and goals

Towards incorporating the ideas expressed in Section 5.1 into our framework, we include a new fixed universe of variables that we call *offset* variables. This universe consists of the denumerable collection $\{Y_1, \ldots, Y_n, \ldots\}$. The variables in this universe can appear in terms and in formulas just as other variables and can appear either bound or free.

We are interested in defining the notion of environment variables for clauses and goals. This is given below.

Definition 5.1. Let C be a clause of the form $\forall x_1 \ldots \forall x_n C'$ such that all of its quantifiers pertain to distinct variables. Then the *environment variables* of C, denoted by $\mathcal{EV}(C)$, are given by $\mathcal{EV}(C) = \{x_1, \ldots, x_n\} \cup \mathcal{F}(C')$. The *environment variables* of a goal G, also written as $\mathcal{EV}(G)$, are given as follows: $\mathcal{EV}(G) = \mathcal{F}(G)$.

We shall be interested in the use of the above definitions mainly in connection with formulas in the earlier syntax, *i.e.*, formulas that do not contain offset variables. However, we will need these definitions, for technical reasons, also for formulas with the revised syntax.

Definition 5.2. An *offset table* is a bijective function from some finite subset of $\mathcal{FV} \cup \mathcal{BV}$ to $\{Y_1, Y_2, \ldots, Y_m\}$ for some number m; the Y_i's are offset variables here. We represent the universe of offset tables by OFFSET TABLE.

An offset table can be viewed as a substitution and we will adopt this viewpoint below.

We now want to consider the preprocessing of goals and clauses formally. For this reason, we introduce modified notions of goal formulas and clauses. As already explained, a clause will now include the following components:

(1) an integer indicating the size of its "environment",
(2) a table indicating initialization for its free variables, and
(3) the head and body of the clause with the environment variables replaced by offset variables.

With regard to (2), based on our informal discussion in Section 5.1, we observe that the free variables in the clause will in general be considered to be offset variables in two different environments. The information for initialization must thus be a pairing of offset variables.

Definition 5.3. An *initialization table* is a one-to-one function from a finite set of offset variables to a finite set of offset variables.

An initialization table may also be thought of as a substitution, and we use this viewpoint at places below.

Definition 5.4. A *preprocessed atomic formula* is like an atomic formula with the difference that the only variables appearing in it are offset variables. We use PPATOM to denote the universe of such preprocessed atomic formulas.

Definition 5.5. Let A denote a preprocessed atomic formula. The category of preprocessed clauses and preprocessed goals, denoted by C and G respectively, are defined by the following syntax rules:

$$G ::= A \mid G \wedge G \mid G \vee G \mid D \supset G$$
$$C ::= \langle N, IT, A\rangle \mid \langle N, IT, G \supset A\rangle$$
$$D ::= [C] \mid C :: D$$

where N represents natural numbers, Y represents offset variables and IT represents initialization tables.

In the above, we expect C of the form $\langle N, IT, G \supset A\rangle$ ($\langle N, IT, A\rangle$) to satisfy the following constraints:

(a) All the free offset variables in $G \supset A$ (A) are included in $\{Y_1, \ldots, Y_N\}$.
(b) The domain of the initialization table is a subset of the free variables appearing in $G \supset A$ (A),

These requirements are expressed by the definition of well-formed preprocessed goals and clauses that is provided below.

Definition 5.6. The notions of *well-formed* preprocessed goals and clauses and of free variables in these goals and clauses are given recursively as follows:

(1) Let G be A, a preprocessed atomic formula. Then G is a well-formed preprocessed goal. In this case, $\mathcal{F}(G) = \{Y_i \mid Y_i \text{ appears in } A\}$.
(2) Let G be $G_1 \wedge G_2$ or $G_1 \vee G_2$. Then G is a well-formed preprocessed goal if G_1 and G_2 are well-formed preprocessed goals. In this case, $\mathcal{F}(G) = \mathcal{F}(G_1) \cup \mathcal{F}(G_2)$.
(3) Let G be $[C_1, \ldots, C_n] \supset G_1$. Then G is a well-formed preprocessed goal if G_1 is a well-formed preprocessed goal and, for $1 \leq i \leq n$, C_i is a well-formed preprocessed clause. In this case, $\mathcal{F}(G) = \mathcal{F}(C_1) \cup \ldots \cup \mathcal{F}(C_n) \cup \mathcal{F}(G_1)$.
(4) Let C be $\langle N, IT, G \supset A\rangle$. Then C is a well-formed preprocessed clause if the following hold:
   (a) G is a well-formed preprocessed goal, and
   (b) $dom(IT) \subseteq (\mathcal{F}(G) \cup \mathcal{F}(A)) \subseteq \{Y_1, \ldots, Y_N\}$.
   In this case, $\mathcal{F}(C) = range(IT)$.
(5) Let C be $\langle N, IT, A\rangle$. Then C is a well-formed preprocessed clause if the following holds: $dom(IT) \subseteq \mathcal{F}(A) \subseteq \{Y_1, \ldots, Y_N\}$. In this case, $\mathcal{F}(C) = range(IT)$.

We assume an extension of the notion of free variables of a preprocessed clause to a *list* of preprocessed clauses as follows: If D is of the form $[C_1, \ldots, C_n]$ where each $C_i, 1 \leq i \leq n$, is a well-formed preprocessed clause, then $\mathcal{F}(D) = \mathcal{F}(C_1) \cup \ldots \cup \mathcal{F}(C_n)$.

Example 5.1. Consider the preprocessed clause of the form $\langle 3, \{\langle Y_4, Y_3\rangle\}, G \supset A\rangle$. This is not well-formed because Y_4 is not an element of $\{\langle Y_1, Y_2, Y_3\rangle\}$.

We henceforth assume that we are dealing only with well-formed preprocessed goals and well-formed preprocessed clauses. The universes of these formulas will be represented by PPCLAUSE and PPGOAL.

The intended correspondence between goals and clauses and their preprocessed versions is made clear by a pair of mappings. The first of these produces a preprocessed version for a given goal or clause and can thus be thought of as the preprocessing function. In the course of defining this map, we will need to "combine" two offset tables to produce an initialization table. This process is defined as follows.

Definition 5.7. If ϑ and ϑ' are two offset tables, then

$$init(\vartheta, \vartheta') = \{\langle Y_i, Y_j\rangle \mid \langle x, Y_i\rangle \in \vartheta \text{ and } \langle x, Y_j\rangle \in \vartheta'\}.$$

The preprocessing functions have the following character: one of them takes a clause and an offset table whose domain includes the free variables in the clause to a preprocessed clause. The other takes a goal and an offset table whose domain includes the environment variables in the goal to a preprocessed goal. These two functions are called Ψ_C and Ψ_G respectively and are defined as follows:

Definition 5.8. The functions

$$\Psi_C : \text{CLAUSE} \times \text{OFFSET TABLE} \rightarrow \text{PPCLAUSE}$$

$$\Psi_G : \text{GOAL} \times \text{OFFSET TABLE} \rightarrow \text{PPGOAL}$$

are such that $\Psi_C(C, \vartheta)$ is defined only if $\mathcal{F}(C) \subseteq dom(\vartheta)$ and all quantifiers in C pertain to distinct variables, and $\Psi_G(G, \vartheta)$ is defined only if $\mathcal{EV}(G) \subseteq dom(\vartheta)$ and all quantifiers in G pertain to distinct variables, and are given, in this case, as follows:

(1) $\Psi_G(A, \vartheta) = \vartheta(A)$
(2) $\Psi_G((C_1 \wedge \ldots \wedge C_n) \supset G, \vartheta) = ([\Psi_C(C_1, \vartheta), \ldots, \Psi_C(C_n, \vartheta)]) \supset \Psi_G(G, \vartheta)$
(3) $\Psi_G(G_1 \wedge G_2, \vartheta) = \Psi_G(G_1, \vartheta) \wedge \Psi_G(G_2, \vartheta)$
(4) $\Psi_G(G_1 \vee G_2, \vartheta) = \Psi_G(G_1, \vartheta) \vee \Psi_G(G_2, \vartheta)$
(5) Let C be $\forall x_1 \ldots \forall x_n(G \supset A)$, let N be the size of the set of environment variables of C and let ϑ' be an offset table for $\mathcal{EV}(C)$. Then $\Psi_C(C, \vartheta) = \langle N, IT, \Psi_G(G, \vartheta') \supset \vartheta'(A)\rangle$ where $init(\vartheta', \vartheta) = IT$.
(6) Let C be $\forall x_1 \ldots \forall x_n A$, let N be the size of the set of environment variables of C and let ϑ' be an offset table for $\mathcal{EV}(C)$. Then $\Psi_C(C, \vartheta) = \langle N, IT, \vartheta'(A)\rangle$ where $init(\vartheta', \vartheta) = IT$.

In the definition above, we assume some fixed but unspecified method of generating an offset table.

The functions Ψ_D and $\Psi_{\mathcal{P}}$ are extensions of Ψ_C to conjunctions and lists of clauses that are defined as follows:

- $\Psi_D(D, \vartheta) = [\Psi_C(C_1, \vartheta), \ldots, \Psi_C(C_n, \vartheta)]$ provided that D is $C_1 \wedge \ldots \wedge C_n$.
- $\Psi_{\mathcal{P}}(\mathcal{P}, \vartheta) = [\Psi_C(C_1, \vartheta), \ldots, \Psi_C(C_n, \vartheta)]$ provided that $\mathcal{P}$ is $[C_1, \ldots, C_n]$.

Example 5.2. Let C be a clause of the form $\forall z_1 \forall z_2 \forall z_3 (q(z_2, z_3, x_1)) \supset p(z_1))$ and let $\vartheta = \{\langle x_1, Y_1 \rangle\}$. Then $\mathcal{EV}(C) = \{z_1, z_2, z_3, x_1\}$. Let C' be such that $C' = \Psi_C(C, \vartheta)$. Then C' might be of the form $\langle 4, \{\langle Y_1, Y_1 \rangle\}, (q(Y_3, Y_4, Y_1)) \supset p(Y_2)\rangle$, and this would be obtained by using an offset table for $\mathcal{EV}(C)$ of the form $\{\langle z_1, Y_2 \rangle, \langle z_2, Y_3 \rangle, \langle z_3, Y_4 \rangle, \langle x_1, Y_1 \rangle\}$.

In defining the mapping in the converse direction, we shall need the notion of a header sequence of a preprocessed clause.

Definition 5.9. Given a preprocessed clause C of the form $\langle N, IT, G \supset A \rangle$ or $\langle N, IT, A \rangle$, the *header sequence* of C, denoted by $hseq(C)$, is defined as a listing of the variables in $\{Y_1, \ldots, Y_N\} - dom(IT)$.

We assume here that the offset variables in the relevant sets are listed in some fixed (but unspecified) order.

Next, we need to define the notion of partially preprocessed formulas. These are given as follows:

Definition 5.10. A *partially preprocessed* clause or goal formula is like a clause or goal with the exception that all the free variables appearing in it are offset variables.

We now define two functions, one for converting a preprocessed clause to a partially preprocessed clause, and the other for converting a preprocessed goal to a partially preprocessed goal.

Definition 5.11. The functions Λ_C and Λ_G are intended to return a partially preprocessed clause and a partially preprocessed goal given a preprocessed clause and a preprocessed goal respectively, and are defined as follows:

(1) $\Lambda_G(A) = A$
(2) $\Lambda_G([C_1, \ldots, C_n]) \supset G) = ([\Lambda_C(C_1), \ldots, \Lambda_C(C_n)]) \supset \Lambda_G(G)$
(3) $\Lambda_G(G_1 \wedge G_2) = \Lambda_G(G_1) \wedge \Lambda_G(G_2)$
(4) $\Lambda_G(G_1 \vee G_2) = \Lambda_G(G_1) \vee \Lambda_G(G_2)$
(5) Let C be $\langle N, IT, G \supset A \rangle$ and let $hseq(C) = [Y_{h_1}, \ldots, Y_{h_s}]$. Then

$$\Lambda_C(C) = IT(\forall z_1 \ldots \forall z_s [z_1/Y_{h_1}] \ldots [z_s/Y_{h_s}](\Lambda_G(G) \supset A))$$

where $z_1, \ldots, z_s$ are variables in $\mathcal{BV}$ which do not appear in $\Lambda_G(G)$.
(6) Let C be $\langle N, IT, A \rangle$ and let $hseq(C) = [Y_{h_1}, \ldots, Y_{h_s}]$. Then

$$\Lambda_C(C) = IT(\forall z_1 \ldots \forall z_s [z_1/Y_{h_1}] \ldots [z_s/Y_{h_s}] A)$$

where $z_1, \ldots, z_s$ are variables in $\mathcal{BV}$.

The desired map from preprocessed clauses and goals to clauses and goals, respectively, is obtained by using the map to partially preprocessed formulas and interpreting the free offset variables relative to a base value. This is made precise below.

Definition 5.12. The functions

$$\Phi_C : \text{PPCLAUSE} \times \mathcal{N} \to \text{CLAUSE}$$
$$\Phi_G : \text{PPGOAL} \times \mathcal{N} \to \text{GOAL}$$

are defined as follows:

(1) $\Phi_G(G, \kappa) = \chi(\Lambda_G(G))$ where $\chi = \{\langle Y_i, w(\kappa+i-1)\rangle \mid Y_i \in \mathcal{F}(\Lambda_G(G))\}$.
(2) $\Phi_C(C, \kappa) = \chi(\Lambda_C(C))$ where $\chi = \{\langle Y_i, w(\kappa+i-1)\rangle \mid Y_i \in \mathcal{F}(\Lambda_C(C))\}$.

The following functions are obvious extensions of the function Φ_C.

- $\Phi_D(D, \kappa) = \Phi_C(C_1, \kappa) \wedge \ldots \wedge \Phi_C(C_n, \kappa)$ where D is $[C_1, \ldots, C_n]$.
- $\Phi_{\mathcal{P}}(\mathcal{P}) = [\Phi_D(D_1, \kappa_1), \ldots, \Phi_D(D_n, \kappa_n)]$ where $\mathcal{P}$ is $[\langle D_1, \kappa_1\rangle, \ldots, \langle D_n, \kappa_n\rangle]$.

Example 5.3. Consider a preprocessed clause $C = \langle 4, \{\langle Y_1, Y_1\rangle\}, (q(Y_3, Y_4, Y_1)) \supset p(Y_2)\rangle$. This preprocessed clause was produced as a result of the preprocessing process in Example 5.2. It is easily seen that $\Lambda_C(C) = \forall z \forall z' \forall z''(q(z', z'', Y_1)) \supset p(z))$ and $\Phi_C(C, \kappa) = \forall z \forall z' \forall z''(q(z', z'', w(\kappa)) \supset p(z))$ for some variables z, z' and z'' in $\mathcal{BV}$.

The following lemma describes some properties of the functions above.

Lemma 4. *Let C be a well-formed preprocessed clause, let C' be the partially preprocessed clause $\Lambda_C(C)$ and let C'' be the clause $\Phi_C(C, \kappa)$. Then it is the case that (i) $\mathcal{F}(C') = \mathcal{F}(C)$, and (ii) $\mathcal{F}(C'') = \{w(\kappa+i-1) \mid Y_i \in \mathcal{F}(C)\}$. Similarly, let G be a well-formed preprocessed goal, let G' be the partially preprocessed goal $\Lambda_G(G)$ and let G'' be the goal $\Phi_G(G, \kappa)$. Then it is the case that (i) $\mathcal{F}(G') = \mathcal{F}(G)$, and (ii) $\mathcal{F}(G'') = \{w(\kappa+i-1) \mid Y_i \in \mathcal{F}(G)\}$.*

Proof. By an induction on the structure of preprocessed clauses and goals.

The following lemma shows that the functions Φ_C and Φ_G are indeed the inverses of Ψ_C and Ψ_G respectively up to π-equivalence.

Lemma 5. *Let C be a clause such that all quantifiers in it pertain to distinct variables and such that $\mathcal{F}(C) = \{x_1, \ldots, x_n\}$. Further, let C' be the preprocessed clause $\Psi_C(C, \vartheta)$ where ϑ is an offset table such that $\vartheta \supseteq \{\langle x_1, Y_{f_1}\rangle, \ldots, \langle x_n, Y_{f_n}\rangle\}$. Finally, let $\gamma = \{\langle w(\kappa + f_i - 1), x_i\rangle \mid 1 \le i \le n\}$ for some natural number κ. Then it is the case that C' is well-formed, $\mathcal{F}(C') = \{Y_{f_1}, \ldots, Y_{f_n}\}$, and $C \equiv_\pi \gamma(\Phi_C(C', \kappa))$. Similarly, let G be a goal formula such that all quantifiers in it pertain to distinct variables and $\mathcal{F}(G) = \{x_1, \ldots, x_n\}$. Let G' be a preprocessed goal $\Psi_G(G, \vartheta)$ where ϑ is an offset table such that $\vartheta = \{\langle x_1, Y_{f_1}\rangle, \ldots, \langle x_n, Y_{f_n}\rangle\}$. Finally, let $\gamma = \{\langle w(\kappa + f_i - 1), x_i\rangle \mid 1 \le i \le n\}$ for some natural number κ. Then it is the case that G' is well-formed, $\mathcal{F}(G') = \{Y_{f_1}, \ldots, Y_{f_n}\}$, and $G \equiv_\pi \gamma(\Phi_G(G', \kappa))$.*

Proof. By an induction on the number of connectives in clauses and goals and also using Lemma 4.

5.3 A Machine Incorporating Environments

To incorporate the scheme outlined in Section 5.1 into $\mathcal{M}_1$, several modifications are necessary.

First, each node needs to maintain a list of environments for the preprocessed clauses that have been used along the path from the root to the node. For this purpose, a new decorating function $\mathcal{E}$ is provided. Now, when a preprocessed clause is used, a new environment (represented by a natural number) for the clause will be created and then added to the front of the existing $\mathcal{E}$ list. When this clause is "solved", the environment for the clause will be removed from the $\mathcal{E}$ list. Thus, the $\mathcal{E}$ list will be maintained in a stack-like manner. In addition, each node must maintain a substitution that records the bindings generated by the INIT rule to be given below. For this reason, a new decorating function φ is introduced.

Secondly, a program, which we shall continue to denote by $\mathcal{P}$, now consists of a list of pairs $\langle D, \kappa \rangle$ where D is a list of preprocessed clauses and κ is the environment relative to which the free variables in D are determined. We shall use the universe CLOSLIST (for closures list) whose elements are lists of such tuples to represent programs in this context.

Finally, the decorated goal sequence associated with each node will take the form of a list of tuples $\langle G, \mathcal{P} \rangle$, interspersed with a marker $\$cls$, where G is a preprocessed goal and $\mathcal{P}$ is a program. The marker $\$cls$ is used to mark the end of the body of a clause instance at which point the environment associated with the clause instance must be "discarded".

These various changes are manifest in the abstract state machine called $\mathcal{M}_2$ that we now present precisely.

5.3.1 Universes and Functions of the Machine $\mathcal{M}_2$

Most universes and functions from $\mathcal{M}_1$ are retained in the new machine, except for those described explicitly below.

(1) The universe of offset variables is added. The universes GOAL, CLAUSE and ATOM are replaced by PPGOAL, PPCLAUSE, and PPATOM described in the previous section. Finally, the universe PROGRAM is replaced by CLOSLIST.

(2) We assume the existence of a universe of markers. This universe consists of $\{\$cls\}$.

(3) The decorating functions associated with nodes are extended and modified as follows:
   - $\mathcal{G}$: NODE $\rightarrow$ (PPGOAL $\times$ CLOSLIST + $\{\$cls\})^*$ is a function that yields, for a given node, a list of decorated goals interspersed with the marker $\$cls$. By an extension of terminology, we shall refer to such a list also as a decorated goal sequence.
   - $\mathcal{E}$: NODE $\rightarrow$ $\mathcal{N}^*$ is a function that yields a list of environments for a given node.
   - φ : NODE $\rightarrow$ SUBST is a function that yields a substitution for a given node. This substitution is distinct from the substitution obtained by using the function θ and maintains the bindings generated by the initialization phase that is to be described.
   - *cands* has the type NODE $\rightarrow$ (PPCLAUSE $\times \mathcal{N})^*$, *candg* has the type NODE $\rightarrow$ PPGOAL*, and i has the type NODE $\rightarrow$ CLOSLIST.

(4) Solving an atomic goal requires clauses whose heads are unifiable with the goal to be selected. The function *procdef* that finds these clauses has a new type: CLOSLIST $\times$ PPATOM $\rightarrow$ (PPCLAUSE $\times$ $\mathcal{N})^*$. Given a program $\mathcal{P}$ and a preprocessed atomic goal A', the function yields a list of pairs of

the form $\langle C, \kappa\rangle$ where C is a preprocessed clause of the form $\langle N, IT, A\rangle$ or $\langle N, IT, G \supset A\rangle$ and κ is the associated environment such that the predicate symbol of A' and its arity are identical to those of A. As before, we expect that the order in which such clauses are listed respects the order in which they appear in $\mathcal{P}$.

(5) A new constant si is introduced to temporarily hold what might be called the *surrounding environment* for a selected clause. In addition, the register cll now has the type PPCLAUSE.

(6) The mode *enter* is refined into two modes: *alloc* and *init*. In *alloc* mode the computation consists of allocating an environment and an initialization of the free variables of the clause is carried out in *init* mode.

5.3.2 The transition system

The initial and final states of $\mathcal{M}_2$ are defined below. We note that an initial state of $\mathcal{M}_2$ is parameterized by the choice of program and (initial) goal.

Definition 5.13. Let $\mathcal{P}$ be a list of well-formed preprocessed clauses and let G be a well-formed preprocessed goal such that $\mathcal{F}(\mathcal{P})$ is an empty set. An *initial state* of $\mathcal{M}_2$ with program $\mathcal{P}$ and goal G is a state in which

(i) *mode* is set to *call* and *stop* is set to 0.

(ii) there are two nodes in NODE and *root* and *cnode* reference these. Further, the contents of these nodes are as follows:

 (a) *root* is the *nil* node, *i.e.*, it is a node on which all the decorating functions are undefined,

 (b) $\mathcal{G}(cnode) = \langle G, [\langle \mathcal{P}, 0\rangle]\rangle :: [\$cls]$, $vi(cnode)$ is set to the size of $\mathcal{EV}(G)$, both $\theta(cnode)$ and $\varphi(cnode)$ are empty substitutions, $\mathcal{E}(cnode)$ is $[0]$, and $b(cnode) = root$.

Definition 5.14. A *final success state* of $\mathcal{M}_2$ with program $\mathcal{P}$ and goal G is a state in which $stop = 1$. In this case, $\theta(cnode) \circ \varphi(cnode)$ restricted to $\{w(i-1) \mid Y_i \in \mathcal{F}(G)\}$ is referred to as the *answer substitution*. A *final failure state* is a state in which $stop = -1$.

We present transition rules of $\mathcal{M}_2$ in Figures 4 – 6. In the presentation of these rules, we shall use the following abbreviations in addition to those described in Section 4:

(1) e shall represent the first element of $\mathcal{E}(cnode)$, y_i shall represent the variable with offset i in the current environment, *i.e.*, the variable $w(e+i-1)$, and ye_i shall represent the variable with offset i in the surrounding environment for a selected clause, *i.e.*, the variable $w(si+i-1)$.

(2) The symbol _ represents a "don't-care" value.

The rules presented are, largely, self explanatory. We observe that, in processing an atomic goal, $\mathcal{M}_2$ utilizes an extra transition rule in comparison with $\mathcal{M}_1$, namely, the DEALLOCATE rule. This rule is used to remove, after a clause is solved, the environment for it.

(1) the AND rule

```
if mode = call & 𝒢 = ⟨G₁ ∧ G₂, 𝒫⟩ :: 𝒢'
then  𝒢 := ⟨G₁, 𝒫⟩ :: ⟨G₂, 𝒫⟩ :: 𝒢'
endif
```

(2) the AUGMENT rule

```
if mode = call & 𝒢 = ⟨D ⊃ G, 𝒫⟩ :: 𝒢'
then  𝒢 := ⟨G, ⟨D, e⟩ :: 𝒫⟩ :: 𝒢'
endif
```

(3) the OR rule

```
if mode = call & 𝒢 = ⟨G₁ ∨ G₂, 𝒫⟩ :: 𝒢'
then  𝒢 := 𝒢'; candg := [G₁, G₂]; i := 𝒫;
      tmode := trygl; mode = try
endif
```

(4) the TRY GOAL rule

```
if mode = try & tmode = trygl
then  case candg of
      []:        backtrack
      G :: Goals:  candg := Goals;
                   extend NODE by t with
                   𝒢(t) := ⟨G, i⟩ :: 𝒢; vi(t) := vi; θ(t) := θ; φ(t) = φ;
                   ℰ(t) = ℰ; b(t) := cnode; cnode := t
                   endextend;
                   mode = call
      endcase
endif
```

Figure 4: Transition rules for solving complex goals

In our analysis of transition sequences we wish not to split the processing of atomic goals into separate parts. This motivates the following definition:

Definition 5.15. The essential states of $\mathcal{M}_2$ are categorized as follows:

- The final success or failure states.
- $stop = 0$ and $mode$ is set to try.
- $stop = 0$ and $mode$ is set to $call$ and $\mathcal{G}(cnode)$ is not of the form $\$cls :: \mathcal{G}'$.

The following theorem is easily established.

Theorem 6. *All initial states and final states are essential states of the interpreter. Furthermore, any transition sequence from an initial state to a final state can be decomposed into a sequence of transitions between essential states.*

We have described a transition system corresponding to $\mathcal{M}_2$, and it is easily observed that the system is well-defined. However, for later analyses, it is useful to categorize each variable introduced to a node η as one of the following:

(1) the SELECTION rule

`if` $mode = call \;\&\; \mathcal{G} = \langle A, \mathcal{P} \rangle :: \mathcal{G}'$
`then` $\mathcal{G} := \mathcal{G}'$; $cands := procdef(A, \mathcal{P})$; $g := \Phi_G(A, e)$; $i := \mathcal{P}$;
$tmode := trycl$; $mode := try$
`endif`

(2) the TRY CLAUSE rule

`if` $mode = try \;\&\; tmode = trycl$
`then` `case` $cands$ `of`
$[]$: $backtrack$
$\langle C, \kappa \rangle :: Cands$: $cands := Cands$;
`extend` NODE `by` t `with`
$\mathcal{G}(t) := \mathcal{G}$; $vi(t) := vi$; $\theta(t) := \theta$; $g(t) := g$; $i(t) := i$;
$\mathcal{E}(t) = \mathcal{E}$; $\varphi(t) = \varphi$; $b(t) := cnode$; $cnode := t$
`endextend`;
$cll := C$; $si := \kappa$; $mode := alloc$
`endif`

(3) the ALLOCATE rule

`if` $mode = alloc$
`then` `let` $cll = \langle N, \_, \_ \rangle$ `in`
$\mathcal{E} := vi :: \mathcal{E}$; $vi := vi + N$; $mode := init$
`endlet`
`endif`

(4) the INITIALIZE rule

`if` $mode = init$
`then` `let` $cll = \langle \_, IT, \_ \rangle$ `and` $IT = [\langle Y_{i_1}, Y_{j_1} \rangle, \ldots, \langle Y_{i_n}, Y_{j_n} \rangle]$ `in`
$\varphi := \varphi \circ \{\langle y_{i_1}, ye_{j_1} \rangle, \ldots, \langle y_{i_n}, ye_{j_n} \rangle\}$; $mode := unify$
`endlet`
`endif`

(5) the UNIFY rule

`if` $mode = unify$
`then` `case` cll `of`
$\langle \_, \_, G \supset A \rangle$: `case` $unify([\langle \theta \circ \varphi(g), \theta \circ \varphi(\Phi_G(A, e)) \rangle])$ `of`
$fail$: $backtrack$
σ : $\mathcal{G} := \langle G, i \rangle :: \$cls :: \mathcal{G}$; $\theta := \sigma \circ \theta$; $mode = call$
`endcase`
$\langle \_, \_, A \rangle$: `case` $unify([\langle \theta \circ \varphi(g), \theta \circ \varphi(\Phi_G(A, e)) \rangle])$ `of`
$fail$: $backtrack$
σ : $\mathcal{G} := \$cls :: \mathcal{G}$; $\theta := \sigma \circ \theta$; $mode = call$
`endcase`
`endcase`
`endif`

Figure 5: Transition rules related to solving atomic goals

(1) the QUERY SUCCESS rule

`if` $mode = call$ & $\mathcal{G} = []$
`then` $stop := 1$
`endif`

(2) the DEALLOCATE rule

`if` $mode = call$ & $\mathcal{G} = \$cls :: \mathcal{G}'$
`then` `let` $\mathcal{E} = e :: \mathcal{E}'$ `in` $\mathcal{E} := \mathcal{E}'$; $\mathcal{G} := \mathcal{G}'$ `endlet`
`endif`

Figure 6: Other rules

(i) as an *auxiliary* variable in that it is introduced for initialization, *i.e.*, it appears in $dom(\varphi(\eta))$,

(ii) as an *essential* variable in the case that it is not an auxiliary variable.

We make these notions precise in the following definitions.

Definition 5.16. Let η be a node in NODE. Then, $\mathcal{V}(\eta)$ denotes the set of variables generated along the path from the root to η, *i.e.*, $\mathcal{V}(\eta) = \{w(0), \ldots, w(vi(\eta) -1)\}$. The set of *essential* variables of η is denoted by $\mathcal{V}_e(\eta)$ and is given by $\mathcal{V}_e(\eta) = \mathcal{V}(\eta) - dom(\varphi(\eta))$.

Below we identify "properness" property that all nodes that arise in a computation will have. In describing this property, it will be convenient to transform goal sequences, candidate clauses and candidate goals that adorn nodes in the machine $\mathcal{M}_2$ into a decoded form. The preprocessing functions defined in Section 5.1 provide the basic ingredients for such a transformation. However, we shall also need the following extensions to these functions.

Definition 5.17. Let $\mathcal{G}$ be a decorated goal sequence and let $\mathcal{E}$ be an environment sequence associated with a node of $\mathcal{M}_2$. Then the function $\Phi_{\mathcal{G}}$ on $\mathcal{G}$ and $\mathcal{E}$ is defined as follows:

- $\Phi_{\mathcal{G}}([], []) = []$
- $\Phi_{\mathcal{G}}(\$cls :: \mathcal{G}, \kappa :: \mathcal{E}) = \Phi_{\mathcal{G}}(\mathcal{G}, \mathcal{E})$
- $\Phi_{\mathcal{G}}(\langle G, \mathcal{P}\rangle :: \mathcal{G}, \kappa :: \mathcal{E}) = \langle \Phi_G(G, \kappa), \Phi_{\mathcal{P}}(\mathcal{P})\rangle :: \Phi_{\mathcal{G}}(\mathcal{G}, \kappa :: \mathcal{E})$

The function $\Phi_{[C]}$ that produces a list of clauses from a list of preprocessed clauses paired with natural numbers is defined as follows:

- $\Phi_{[C]}([]) = []$
- $\Phi_{[C]}(\langle C, \kappa\rangle :: Cands) = \Phi_C(C, \kappa) :: \Phi_{[C]}(Cands)$

Finally, the function $\Phi_{[G]}$ that produces a list of goals from a list of preprocessed goals and a list of environments is defined as follows:

- $\Phi_{[G]}([], \_) = []$
- $\Phi_{[G]}(G :: Goals, \kappa :: \mathcal{E}) = \Phi_G(G, \kappa) :: \Phi_{[G]}(Goals, \kappa :: \mathcal{E})$

Definition 5.18. A node η is said to be *proper* if it is the *root* node, or the following hold:

(i) Every preprocessed formula that appears in $\mathcal{G}(\eta)$ is well-formed,
(ii) $range(\varphi(\eta))$, $dom(\theta(\eta))$ and $range(\theta(\eta))$ are subsets of $\mathcal{V}_e(\eta)$,
(iii) The set of free variables appearing $\Phi_{\mathcal{G}}(\mathcal{G}(\eta), \mathcal{E}(\eta))$ is a subset of $\mathcal{V}(\eta)$,
(iv) If $tmode(\eta) = trycl$, then the set of free variables appearing in $\Phi_{\mathcal{P}}(i(\eta))$, $\Phi_{[C]}(cands(\eta))$ or $g(\eta)$ is a subset of $\mathcal{V}(\eta)$, and
(v) If $tmode(\eta) = trygl$, then the set of free variables appearing in $\Phi_{\mathcal{P}}(i(\eta))$ or $\Phi_{[G]}(candg(\eta), \mathcal{E}(\eta))$ is a subset of $\mathcal{V}(\eta)$.

An essential state is said to be *proper* if every node in it is proper.

Lemma 7. *Each essential state in $\mathcal{M}_2$ with program $\mathcal{P}$ and goal G is proper.*

Proof. By induction on the number of transitions leading to an essential state. The conditions for properness follow from a simple inspection of the transition rules, utilizing the well-formedness of $\mathcal{P}$ and G and possibly also Lemma 4.

In the next section, we verify that this machine is equivalent to the machine $\mathcal{M}_1$.

6 Equivalence of the Machines $\mathcal{M}_1$ and $\mathcal{M}_2$

Our interest in this section is to show that $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent from the perspective of the answer substitution that is computed. These answers will in general not be identical. However, we will show that they can be made identical by renaming free variables in a consistent fashion and we consider this sufficient to claim equivalence. This is made precise in the following definition.

Definition 6.1. Two answer substitutions θ_1 and θ_2 are considered *equivalent* if there is a bijection γ from $\mathcal{F}(\theta_2)$ to $\mathcal{F}(\theta_1)$ such that $\theta_1 \circ \gamma = \gamma \circ \theta_2$.

In showing the equivalence between $\mathcal{M}_1$ and $\mathcal{M}_2$, we assume this notion of equivalence for answer substitutions. Keeping this in mind, we define the following notion of correspondence between states of $\mathcal{M}_2$ and $\mathcal{M}_1$.

Definition 6.2. A node η_2 in the machine $\mathcal{M}_2$ with a list of preprocessed clauses $\mathcal{P}$ and a preprocessed goal G *Φ-corresponds* to a node η_1 in the machine $\mathcal{M}_1$ with program $\mathcal{P}'$ and goal G' if both η_2 and η_1 are the *root* node in the two machines or if the following conditions are satisfied:

(1) $b(\eta_2)$ Φ-corresponds to $b(\eta_1)$
(2) There exists a bijection γ from $\mathcal{V}_e(\eta_2)$ to $\mathcal{V}(\eta_1)$ such that
  (a) γ is also a bijection from $\mathcal{F}(G)$ to $\mathcal{F}(G')$,
  (b) $\theta(\eta_1) \circ \gamma = \gamma \circ \theta(\eta_2)$, *i.e.*, γ commutes with the application of substitutions in $\mathcal{M}_2$ and $\mathcal{M}_1$,
  (c) $\mathcal{G}(\eta_1) = \gamma \circ \varphi(\eta_2)(\Phi_{\mathcal{G}}(\mathcal{G}(\eta_2), \mathcal{E}(\eta_2)))$, *i.e.*, the decoded decorated goal sequence in η_2 and the decorated goal sequence in η_1 correspond up to the renaming contained in γ,

(d) If $tmode(\eta_2) = trycl$, then $tmode(\eta_1) = trycl$, and (i) $g(\eta_1) = \gamma \circ \varphi(\eta_2)(g(\eta_2))$, (ii) $i(\eta_1) = \gamma \circ \varphi(\eta_2)(\Phi_{\mathcal{P}}(i(\eta_2)))$, and (iii) $cands(\eta_1) = \gamma \circ \varphi(\eta_2)(\Phi_{[C]}(cands(\eta_2)))$, *i.e.*, both nodes are the objects of the SELECTION rule in the two machines, and the current goals, current clauses and candidate clauses correspond up to the renaming contained in γ, and

(e) If $tmode(\eta_2) = trygl$, then $tmode(\eta_1) = trygl$ and (i) $i(\eta_1) = \gamma \circ \varphi(\eta_2)(\Phi_{\mathcal{P}}(i(\eta_2)))$, and (ii) $candg(\eta_1) = \gamma \circ \varphi(\eta_2)(\Phi_{[G]}(candg(\eta_2), \mathcal{E}(\eta_2)))$, *i.e.*, both nodes are the objects of the OR rule in the two machines, and the current clauses and candidate goals correspond up to the renaming contained in γ.

The content of Definition 6.2 should, for the most part, be clear from the preceding discussions. The condition (2b) asserts the equivalence of substitutions associated with these states relative to a renaming of variables defined in Definition 6.1.

Definition 6.3. Let $\mathcal{B}$ be an essential state of $\mathcal{M}_2$ and let $\mathcal{A}$ be an essential state of $\mathcal{M}_1$. Let us refer to the current nodes of $\mathcal{B}$ and $\mathcal{A}$ as $cnode_2$ and $cnode_1$ respectively. Then $\mathcal{B}$ Φ-corresponds to $\mathcal{A}$ if both are the final states in the two machines or *mode* of $\mathcal{B}$ is identical to *mode* of $\mathcal{A}$ and $cnode_2$ Φ-corresponds to $cnode_1$.

Now we claim that the state map Φ is preserved between initial states.

Lemma 8. *Let $\mathcal{P}$ be a list of preprocessed clauses such that $\mathcal{F}(\mathcal{P}) = \emptyset$ and G be a preprocessed goal such that $\mathcal{F}(G) = \{Y_{i_1}, \ldots, Y_{i_r}\}$. Further, let $\mathcal{P}'$ be the program $\gamma(\Phi_{\mathcal{P}}([\langle \mathcal{P}, 0\rangle]))$ and G' be the goal $\gamma(\Phi_G(G, 0))$ where γ is a substitution which is bijective from $\{w(i_1{-}1), \ldots, w(i_r{-}1)\}$ to $\{w(0), \ldots, w(r{-}1)\}$. Then the initial state of $\mathcal{M}_2$ with program $\mathcal{P}$ and goal G Φ-corresponds to the initial state of $\mathcal{M}_1$ with program $\mathcal{P}'$ and goal G' and vice versa.*

Proof. Let us refer to the current nodes of the initial state of $\mathcal{M}_2$ and $\mathcal{A}$ as $cnode_2$ and $cnode_1$ respectively. From Lemma 4, it follows that $\mathcal{F}(G') = \{w(0), \ldots, w(r-1)\}$. Furthermore, it is easily seen that $\mathcal{V}_e(cnode_2) = \{w(i_1 - 1), \ldots, w(i_r - 1)\}$. It follows from this that γ is bijective from $\mathcal{V}_e(cnode_2)$ to $\mathcal{V}(cnode_1)$. The rest of the conditions are straightforward to verify.

The following lemma shows that the notion of Φ-correspondence is preserved between intermediate states.

Lemma 9. *Let $\mathcal{A}$ be an essential state of $\mathcal{M}_1$ that is not a final state and let $\mathcal{B}$ be an essential state of $\mathcal{M}_2$ that is not a final state. If $\mathcal{B}$ Φ-corresponds to $\mathcal{A}$, and if $\mathcal{A}'$ and $\mathcal{B}'$ are the next essential states that $\mathcal{M}_1$ and $\mathcal{M}_2$ transit to from $\mathcal{A}$ and $\mathcal{B}$ respectively, then $\mathcal{B}'$ Φ-corresponds to $\mathcal{A}'$.*

Proof. We prove this lemma by considering the possibilities for $\mathcal{B}$, an essential but not final state in $\mathcal{M}_2$. We refer below to the *cnode* of $\mathcal{B}$ and $\mathcal{A}$ at the outset as η_2 and η_1 respectively.

Let us first consider the case where *mode* of $\mathcal{B}$ is set to *call*. The transition from $\mathcal{B}$ to $\mathcal{B}'$ is based on four different transition rules depending on the value of

$\mathcal{G}(\eta_2)$ and these are the AND, AUGMENT, OR and SELECTION rules. Using the definition of Φ-correspondence, it can be easily verified that *mode* of $\mathcal{A}$ is also set to *call* and $\mathcal{A}$ transits to $\mathcal{A}'$ using the transition rule of the same name in $\mathcal{M}_1$ and $\mathcal{B}'$ Φ-corresponds to $\mathcal{A}'$.

Let us consider the case where *mode* is set to *try* and $tmode(\eta_2)$ is set to *trygl* in $\mathcal{B}$. From the hypothesis, we observe that *mode* is set to *try* and $tmode(\eta_1)$ is set to *trygl* in $\mathcal{A}$. The next essential state, denoted by $\mathcal{B}'$, in $\mathcal{M}_2$ results from $\mathcal{B}$ by using TRY GOAL. Again, from the definition of Φ-correspondence, it is straightforward to verify that $\mathcal{A}$ transits to $\mathcal{A}'$ using TRY GOAL and $\mathcal{B}'$ Φ-corresponds to $\mathcal{A}'$.

Let us consider the case where *mode* is set to *try* and $tmode(\eta_2)$ is set to *trycl* in $\mathcal{B}$. From the hypothesis, we observe that *mode* is set to *try* and $tmode(\eta_1)$ is set to *trycl* in $\mathcal{A}$. In the case of interest, we know that

$$cands(\eta_1) = \gamma \circ \varphi(\eta_2)(\Phi_{[C]}(cands(\eta_2)))$$

where γ is the substitution by virtue of which η_2 Φ-corresponds to η_1. Now, there are two kinds of transitions possible from $\mathcal{B}$ depending on the value of $cands(\eta_2)$. The first kind corresponds to the situation where $cands(\eta_2)$ is an empty list. From the definition of Φ-correspondence, it follows that $cands(\eta_1)$ is also an empty list. Since $\mathcal{B}$ is not a final state, $\mathcal{M}_2$ will transit to a new essential state with *mode* set to *try* and $cnode_2$ set to $b(\eta_2)$, or to the final failure state if $b(\eta_2) = root$. The machine $\mathcal{M}_1$ will also transit to a new essential state with *mode* set to *try* and $cnode_1$ set to $b(\eta_1)$ or to the final failure state if $b(\eta_1) = root$. By definition, $b(\eta_2)$ Φ-corresponds to $b(\eta_1)$ and hence, in either case, the new essential states in the two machines also Φ-correspond.

The other possibility is for $cands(\eta_2)$ to be of the form $\langle C_2, \kappa\rangle :: Cs_2$ where κ is a variable index. Then $cands(\eta_1)$ must be of the form $C_1 :: Cs_1$ where

$$C_1 = \gamma \circ \varphi(\eta_2)(\Phi_C(C_2, \kappa))$$

and

(1) $\quad Cs_1 = \gamma \circ \varphi(\eta_2)(\Phi_{[C]}(Cs_2))$

Now C_2 is either of the form $\langle N, IT, G_2 \supset A_2\rangle$ or of the form $\langle N, IT, A_2\rangle$. We verify the claim only for the former case, since the argument for the latter case is similar but simpler.[3] Let $hseq(C_2)$ be $[Y_{h_1}, \ldots, Y_{h_s}]$, let ye_j denote the variable $w(\kappa+j-1)$ and let $\rho_I = \{\langle Y_i, ye_j\rangle \mid \langle Y_i, Y_j\rangle \in IT\}$. From the definition of Φ_C, it follows easily that C_1 is of the form $\forall z_1 \ldots \forall z_s(G_1 \supset A_1)$ where

(2) $\quad G_1 = [z_1/Y_{h_1}] \ldots [z_s/Y_{h_s}] \circ \gamma \circ \varphi(\eta_2) \circ \rho_I(\Lambda_G(G_2))$, and
(3) $\quad A_1 = [z_1/Y_{h_1}] \ldots [z_s/Y_{h_s}] \circ \gamma \circ \varphi(\eta_2) \circ \rho_I(A_2)$.

Let u_i denote the variable $w(vi(cnode_1)+i-1)$ and let y_i denote the variable $w(vi(cnode_2)+i-1)$. Further, let ρ be the renaming substitution $\{\langle z_i, u_i\rangle \mid 1 \le i \le s\}$, let $\gamma' = \gamma \cup \{\langle y_{h_1}, u_1\rangle, \ldots, \langle y_{h_s}, u_s\rangle\}$ and let $\varphi' = \varphi(\eta_2) \circ \{\langle y_i, ye_j\rangle \mid \langle Y_i, Y_j\rangle \in IT\}$.

We now claim the following:

[3] It is relevant to observe that in the latter case where C_2 is of the form $\langle N, IT, A_2\rangle$, the transition to the next essential state in $\mathcal{M}_2$ might involve a sequence of uses of the DEALLOCATE rule. However, the treatment of this rule is straightforward as the reader may well verify.

(i) $\rho(G_1) = \gamma' \circ \varphi'(\Phi_G(G_2, vi(\eta_2)))$ and $\rho(A_1) = \gamma' \circ \varphi'(\Phi_G(A_2, vi(\eta_2)))$,
(ii) $\theta(\eta_1) \circ \rho(A_1) = \gamma' \circ \theta(\eta_2) \circ \varphi'(\Phi_G(A_2, vi(\eta_2)))$,
(iii) $\theta(\eta_1)(g(\eta_1)) = \gamma \circ \theta(\eta_2) \circ \varphi'(g(\eta_2))$,

Item (i) above follows from (2) and (3) by a reconfiguration of the various substitutions involved; the details are obvious even if somewhat tedious to describe explicitly. Item (ii) follows from (i) by noting that $\theta(\eta_1) \circ \gamma' = \gamma' \circ \theta(\eta_2)$; this observation itself follows from condition (2b) of Definition 6.2 and by noting that the variables in $\{u_1, \ldots, u_s\}$ do not appear in $dom(\theta(\eta_1))$ and, likewise, the variables in $\{y_{h_1}, \ldots, y_{h_s}\}$ do not appear in $dom(\theta(\eta_2))$. For item (iii), we observe first that by Definition 6.2 that $g(\eta_1) = \gamma \circ \varphi(\eta_2)(g(\eta_2))$. But then from condition (2b) of Definition 6.2

$$\theta(\eta_1)(g(\eta_1)) = \gamma \circ \theta(\eta_2) \circ \varphi(\eta_2)(g(\eta_2)).$$

But by the properness of η_2, we have that $\varphi(\eta_2)(g(\eta_2)) = \varphi'(g(\eta_2))$ and hence that

$$\theta(\eta_1)(g(\eta_1)) = \gamma \circ \theta(\eta_2) \circ \varphi'(g(\eta_2)).$$

Now let us inspect the states of both machines right before the UNIFY rule is invoked. This state would be reached in $\mathcal{M}_2$ by invoking in turn the TRY, the ALLOCATE and the INITIALIZE rules. Let t_2 be the new *cnode* in $\mathcal{M}_2$ at the end of this sequence. Similarly, $\mathcal{M}_1$ reaches such a state by executing the TRY and then the ENTER rule. Let t_1 be the new *cnode* in $\mathcal{M}_1$ at the end of this sequence.

We note first that as a result of the transitions caused by the TRY rules in the two machines, η_2 and η_1 are modified so that $cands(\eta_2) = Cs_2$ and $cands(\eta_1) = Cs_1$. Using (1), it follows easily that the resulting η_2 Φ-corresponds to the resulting η_1. We also note the following with regard to the nodes t_2 and t_1:

(v) $\mathcal{G}(t_2)$ is set to $\mathcal{G}(\eta_2)$, $vi(t_2)$ is set to $vi(\eta_2)+N$, $\theta(t_2)$ is set to $\theta(\eta_2)$, $\varphi(t_2)$ is set to φ', $\mathcal{E}(t_2)$ is set to $vi(\eta_2) :: \mathcal{E}(\eta_2)$ and $b(t_2)$ is set to η_2. Furthermore, $g(t_2)$ is set to $g(\eta_2)$, and $i(t_2)$ is set to $i(\eta_2)$.
(vi) $\mathcal{G}(t_1)$ is set to $\mathcal{G}(\eta_1)$, $vi(t_1)$ is set to $vi(\eta_1) + s$, $\theta(t_1)$ is set to $\theta(\eta_1)$. Furthermore, $g(t_1)$ is set to $g(\eta_1)$, and $i(t_1)$ is set to $i(\eta_1)$.

The next step in both machines is to invoke the UNIFY rule. Let B be $\theta(t_2) \circ \varphi(t_2)(g(t_2))$ and let B' be $\theta(t_2) \circ \varphi(t_2)(\Phi_G(A_2, vi(\eta_2)))$. Then $\mathcal{M}_2$ will invoke the unification procedure on $\{\langle B, B'\rangle\}$. Similarly, if A is $\theta(t_1)(g)$ and A' is $\theta(t_1)(\rho(A_1))$, then $\mathcal{M}_1$ invokes the unification procedure on $\{\langle A, A'\rangle\}$. Now we claim the following:

(a) If B and B' have a most general unifier σ_2, then A and A' have a most general unifier σ_1 such that $\sigma_1(y) = \gamma' \circ \sigma_2(x)$ for each $\langle x, y\rangle \in \gamma'$.
(b) If B' and B'' have no unifier, then neither do A' and A''.

To see this, using (ii)-(iii) above, we observe that $A = \gamma'(B)$ and $A' = \gamma'(B')$. Thus $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively invokes the unification procedure on a pair of terms that are identical up to a variable renaming.

Now suppose B and B' have no unifiers. Since $\mathcal{B}$ is not a final state, $\mathcal{M}_2$ will transit to a new essential state with *mode* set to *try* and *cnode* set to η_2. By (b) above, $\mathcal{M}_1$ will also transit to a new essential state with *mode* set to *try* and $cnode_1$ set to η_1. We have already noted that η_2 Φ-corresponds to η_1 and so the lemma follows in this case.

Now suppose B and B' have a most general unifier σ_2. Then, by virtue of (a) above, A and A' have a most general unifier σ_1. Thus, using the definition of the UNIFY rule in the two machines, we see that $\mathcal{M}_2$ and $\mathcal{M}_1$ will transit to new essential states in which *mode* is *call* and with *cnode* t'_2 and t'_1 respectively, that have the following characteristics:

(vii) t'_2 is identical to t_2 except that $\mathcal{G}(t'_2)$ is set to $\langle G_2, i(t_2)\rangle :: \$cls :: \mathcal{G}(t_2)$, and $\theta(t'_2)$ is set to $\sigma_2 \circ \theta(t_2)$. Further, *mode* is set to *call*.

(viii) t'_1 is identical to t_1 except that $\mathcal{G}(t'_1)$ is set to $\langle \rho(G_1), i(t_1)\rangle :: \mathcal{G}(t_1)$, and $\theta(t'_1)$ is set to $\sigma_1 \circ \theta(t_1)$. Furthermore, *mode* is set to *call*.

Our requirement, then, is to verify that t'_2 Φ-corresponds to t'_1. We have already seen that condition (1) of Definition 6.2 is true: $b(t'_2)$ is η_2 and $b(t'_1)$ is η_1. To verify condition (2), we observe first that γ' is a bijection from $\mathcal{V}_e(t'_2)$ to $\mathcal{V}(t'_1)$. This follows from noting that

$$\begin{aligned}\mathcal{V}_e(t'_2) &= \mathcal{V}_e(\eta_2) \cup \{y_1, \ldots, y_N\} - \{y_i \mid Y_i \in dom(\varphi')\} \\ &= \mathcal{V}_e(\eta_2) \cup \{y_{h_1}, \ldots, y_{h_s}\}\end{aligned}$$

and $\mathcal{V}(t'_1) = \mathcal{V}(\eta_1) \cup \{u_1, \ldots, u_s\}$. Now, condition (2a) is obviously true. Condition (2b) follows from (a) above and by noting that that $\theta(\eta_1) \circ \gamma = \gamma \circ \theta(\eta_2)$. Finally condition (2c) follows from (i) and by noting that $\mathcal{G}(\eta_1) = \gamma' \circ \varphi(t'_2)(\Phi_{\mathcal{G}}(\mathcal{G}(\eta_2), \mathcal{E}(\eta_2)))$, and $i(\eta_1) = \gamma' \circ \varphi(t'_2)(\Phi_{\mathcal{P}}(i(\eta_2)))$. Thus we have verified the lemma in this case.

The following lemma is immediate from the definition of Φ-correspondence.

Lemma 10. *Let $\mathcal{A}$ be an essential state of $\mathcal{M}_1$ and let $\mathcal{B}$ be an essential state of $\mathcal{M}_2$ such that $\mathcal{B}$ Φ-corresponds to $\mathcal{A}$. Then $\mathcal{A}$ is a final success (failure) state of $\mathcal{M}_1$ if and only if $\mathcal{B}$ is a final success (failure) state of $\mathcal{M}_2$.*

Lemma 11. *Let $\mathcal{P}$ and G be a list of preprocessed clauses and preprocessed goal such that $\mathcal{F}(G) = \{Y_{i_1}, \ldots, Y_{i_r}\}$, and let $\mathcal{P}'$ be the program $\gamma(\Phi_{\mathcal{P}}([\langle \mathcal{P}, 0\rangle]))$ and G' be the goal $\gamma(\Phi_G(G, 0))$ where γ is a substitution which is bijective from $\{w(i_1 - 1), \ldots, w(i_r - 1)\}$ to $\{w(0), \ldots, w(r-1)\}$. Then the machine $\mathcal{M}_2$ with program $\mathcal{P}$ and goal G is equivalent to the machine $\mathcal{M}_1$ with program $\mathcal{P}'$ and goal G' in the sense of Section 2.2.*

Proof. To show that $\mathcal{M}_1$ simulates $\mathcal{M}_2$, we need to verify that the requirements (a)–(c) of Section 2.2 hold, assuming that $\mathcal{M}_2$ is M', $\mathcal{M}_1$ is M and the notion of Φ-correspondence defined in this section is the map Φ. Lemma 8 and 10 ensure that condition (a) holds. Theorems 2 and 6 ensure that condition (b) holds. Condition (c) follows from Lemma 9. Finally, the reverse direction follows from Lemma 8–10, the fact that Φ is bijective from the essential states of $\mathcal{M}_2$ to the essential states of $\mathcal{M}_1$, and the fact that $\mathcal{M}_1$ is deterministic. Note that our notion of equivalence is based on considering identical two answer substitutions that are equivalent in the sense of Definition 6.1.

The following theorem establishes the main result of this paper.

Theorem 12. *Let $\mathcal{P}'$ and G' be a program and goal, and let $\mathcal{P}$ be the list of preprocessed clauses $\Psi_{\mathcal{P}}(\mathcal{P}', nil))$ and G be the goal $\Psi_G(G', \vartheta)$ where ϑ is an offset table for the environment variables of G'. Then the machine $\mathcal{M}_2$ with program $\mathcal{P}$ and goal G is equivalent to the machine $\mathcal{M}_1$ with program $\mathcal{P}'$ and goal G'.*

Proof. let $\mathcal{P}''$ be a program such that $\mathcal{P}'' = \gamma(\Phi_{\mathcal{P}}([\langle\mathcal{P}, 0\rangle]))$ and let G'' be a goal such that $G'' = \gamma(\Phi_G(G, 0))$. From Lemma 11, it follows that $\mathcal{M}_2$ with $\mathcal{P}$ and G is equivalent to $\mathcal{M}_1$ with $\mathcal{P}''$ and G''. By virtue of Lemma 3, $\mathcal{M}_2$ with $\mathcal{P}$ and G is equivalent to $\mathcal{M}_1$ with any π-variant of $\mathcal{P}''$ and any π-variant of G''. However, it follows from Lemma 5 that $\mathcal{P}'$ is a π-variant of $\mathcal{P}''$ and G' is a π-variant of G''. Thus we have verified the theorem.

Theorem 12, the main result of this paper, observes the correctness of both our preprocessing functions for programs and goals, and the machine incorporating the notion of an environment that is based on it.

7 Conclusion

Our main objective in this paper has been in verifying the correctness of a compilation method for an extension to Prolog with embedded implications. We have achieved this goal by showing that the deterministic interpreter $\mathcal{M}_1$ for the language is equivalent to its refinement $\mathcal{M}_2$ that adopts a closure-based representation of clauses. This closure-based representation separates an instance of a clause with possible free variables into a skeleton part that is fixed during computation and an environment that maintains the part that is dynamically determined. This representation can be easily adapted to other languages with similar features.

The machine $\mathcal{M}_2$ is interesting in that it provides a basis for sharing of structure and for efficient implementations of the language. Thus we can obtain a reasonable abstract machine from $\mathcal{M}_2$ by applying a series of refinements. One particular concern with $\mathcal{M}_2$ is that each goal needs to carry its own program context. This is rather cumbersome and is avoided by managing the program context in a stack-based manner. Hence, a goal such as $D \supset G$ is solved by adding D to the program context, solving G, and then removing D from the program context. In [Nadathur et al. 95], we introduced an *implication* stack to permit the incremental addition and subsequent retraction of clauses from a program context that is needed in an implementation of this approach. However, backtracking may require reinstating states previously in existence and bookkeeping devices are needed to implement this efficiently. We incorporated an efficient bookkeeping mechanism into our model by embedding the implication stack and the stack of choice points into a single stack and by managing them on a chronological basis. We refer the reader to [Nadathur et al. 95] for further details.

Besides the work in [Nadathur et al. 95], there have been another proposal related to the implementation of scoped program clauses: Lamma, Mello and Natali in [Lamma et al. 92] presented, in the extended context of the WAM, a stack-based approach to manage the dynamically changing program. Their proposal is quite similar to the one in [Nadathur et al. 95], but they concerned just program clauses, not closures.

Finally, the ideas presented in this paper are used in the development of an abstract machine for λProlog [Nadathur and Miller 88] — a superset of the language considered here.

References

[Blakley 92] Bob Blakley. *A Smalltalk Evolving Algberas and Its Uses.* PhD thesis, University of Michigan, 1992.

[Börger 90a] Egon Börger. A logical operational semantics of full Prolog. part I. selection core and control. In H. Kleine Büning E. Börger and M. Richter, editors, *3rd Workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Artificial Intelligence*, pages 36–64. Springer-Verlag, 1990.

[Börger 90b] Egon Börger. A logical operational semantics of full Prolog. part II. built-in predicates for database manipulations. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Artificial Intelligence*, pages 1–14. Springer-Verlag, 1990.

[Börger and Rosenzweig 94] Egon Börger and Dean Rosenzweig. The WAM — definition and compiler correctness. In L. C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence. North–Holland, 1994.

[Gabbay and Reyle 84] D. M. Gabbay and U. Reyle. N-Prolog: An extension of Prolog with hypothetical implications. I. *Journal of Logic Programming*, 1:319 – 355, 1984.

[Giordano et al. 88] L. Giordano, A. Martelli, and G. F. Rossi. Local definitions with static scope rules in logic languages. In *Proceedings of the FGCS International Conference, Tokyo*, 1988.

[Gurevich 91] Yuri Gurevich. Evolving algebras. A tutorial introduction. *Bulletin of the European Association for Theoretical Computer Science*, 43:264–284, 1991.

[Gurevich and Morris 88] Yuri Gurevich and James M. Morris. Algebraic operational semantics and Modula-2. In H. Kleine E. Börger and M. Richter, editors, *1st Workshop on Computer Science Logic*, volume 329 of *Lecture Notes in Computer Science*, pages 81–101. Springer-Verlag, 1988.

[Gurevich and Moss 90] Yuri Gurevich and Larry Moss. Algebraic operational semantics and Occam. In H. Kleine E. Börger and M. Richter, editors, *3rd Workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*, pages 176–192. Springer-Verlag, 1990.

[Hodas and Miller 94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, May 1994.

[Kwon 94] Keehang Kwon. *Towards a Verified Abstract Machine for a Logic Programming Language with a Notion of Scope.* PhD thesis, Duke University, December 1994. Also available as Technical Report CS–1994–36 from Department of Computer Science, Duke University.

[Lamma et al. 92] Evelina Lamma, Paola Mello, and Antonio Natali. An extended Warren abstract machine for the execution of structured logic programs. *Journal of Logic Programming*, 14:187–222, 1992.

[McCarty 88a] L. Thorne McCarty. Clausal intuitionistic logic I. Fixed point semantics. *Journal of Logic Programming*, 5(1):1–31, 1988.

[McCarty 88b] L. Thorne McCarty. Clausal intuitionistic logic II. Tableau proof procedures. *Journal of Logic Programming*, 5:93–132, 1988.

[Miller 89] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[Miller 94] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994. IEEE Computer Society Press.

[Miller et al. 91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Miller et al. 87] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In David Gries, editor, *Symposium on Logic in Computer Science*, pages 98–105, Ithaca, NY, June 1987.

[Monteiro and Porto 89] Luís Monteiro and António Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Sixth International Logic Programming Conference*, pages 284–299, Lisbon, Portugal, June 1989. MIT Press.

[Nadathur 93] Gopalan Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11:115–145, 1993.

[Nadathur et al. 95] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, 1995.

[Nadathur and Miller 88] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[Warren 83] D.H.D. Warren. An abstract Prolog instruction set. Technical report, SRI International, October 1983. Technical Note 309.

Acknowledgements

This paper is based on the author's doctoral dissertation. The author would like to thank Gopalan Nadathur for guidance and helpful comments. This research was supported in part by NSF Grants CCR-89-05825, CCR-92-08465 and 1996 University Basic Research Grant from the Korea Ministry of Information and Communications.