LEXICAL SCOPING AS UNIVERSAL QUANTIFICATION

Dale Miller
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104–6389 USA

Abstract: A universally quantified goal can be interpreted intensionally, that is, the goal $\forall x.G(x)$ succeeds if for some new constant c, the goal $G(c)$ succeeds. The constant c is, in a sense, given a scope: it is introduced to solve this goal and is "discharged" after the goal succeeds or fails. This interpretation is similar to the interpretation of implicational goals: the goal $D \supset G$ should succeed if when D is assumed, the goal G succeeds. The assumption D is discharged after G succeeds or fails. An interpreter for a logic programming language containing both universal quantifiers and implications in goals and the body of clauses is described. In its non-deterministic form, this interpreter is sound and complete for intuitionistic logic. Universal quantification can provide lexical scoping of individual, function, and predicate constants. Several examples are presented to show how such scoping can be used to provide a Prolog-like language with facilities for local definition of programs, local declarations in modules, abstract data types, and encapsulation of state.

Appears in the Proceedings of the Sixth International Conference on Logic Programming, Lisboa, Portugal, 19–23 June 1989. Address correspondence to Miller at "dale@linc.cis.upenn.edu" or at the address above.

1. Introduction

In [9], first-order Horn clause programs were extended by allowing implications in the body of clauses and in goals (queries). That extended logic was used to provide a simple and dynamic notion of modular logic programming. This paper extends the logic presented in that paper by permitting universal quantification as well as implications in goals and the body of clauses. The addition of such universal quantifiers strengthen the modular program constructions described in [9] since it makes it possible to provide scope to individual, function, and predicate constants.

The logic described in this paper is related to logics considered by many researchers in logic programming and, most recently, in theorem proving and type theory. See [3, 5, 7, 8, 14] for the description of closely logics applied to logic programming. Similar logics, especially higher-order versions, have been used as meta languages in specifying and implementing theorem provers [2, 18, 19]. The logic presented here is most closely related to the *first-order hereditary Harrop formulas* presented in [14]: it differs only in that we shall provide for more liberal forms of universal quantification in the body of program clauses. This logic, as well as several other extension to Horn clauses, are part of the experimental logic programming language λProlog [17]. The examples in this paper were developed and tested using the LP2.7 [13] and the eLP [1] implementations of λProlog.

Although the scoping concepts described in this paper follow naturally from simple proof-theoretical considerations, the resulting notions of "module" and "abstract datatype" differ significantly from those notions found in other programming languages. In our setting, logic programs defined in a given modules are not necessarily closed: the meaning of the programs defined in them may depend on the context in which they are used. Similarly, the mechanism for supplying security in abstract datatypes is described as a "runtime check"; it cannot, in general, be done at compile time. For proposals of more static notions of modules and abstract datatypes for logic programs, see [4, 15, 20, 21].

2. The logic programming language $\mathcal{L}$

Consider a logic that contains constants and variables for individuals, functions, and predicates. Let A, D, G be syntactic variables that range over the following classes of formulas.

$$A := \text{ atomic formula}$$

$$D := A \mid G \supset A \mid D_1 \wedge D_2 \mid \forall x\ D$$
$$G := A \mid D \supset G \mid G_1 \wedge G_2 \mid \forall y\ G$$

The universal quantifier $\forall xD$ is over individuals only, while the universal quantifier $\forall yG$ is over individuals, functions, and predicates. Let $\mathcal{D}$ be the set of D-formulas and let $\mathcal{G}$ be the set of G-formulas that, in both cases, do not contain free function or predicate variables. The role of free function and predicate variables is restricted only to the construction of D-formulas and G-formulas. During the interpretation of this logic (see Section 4) the only free variables that need to be considered are those that are individual variables. A formula in $\mathcal{G}$ is a *goal* or *query*. A formula in $\mathcal{P}$ is a *definite clause* or *program clauses*, and a finite subset of $\mathcal{P}$ is a *program*.

While this language is not, strictly speaking, first-order, it is far from having the complex meta theory or theorem proving problems associated with higher-order logics and logic programming languages (accounts of which are in [12, 14, 16]). As we shall show, a constrained form of first-order unification makes it possible to implement complete theorem provers and interpreters for $\mathcal{L}$.

Simple modifications of the proof theory discussions in [9] show that $\mathcal{P} \vdash_I G$ (where $\vdash_I$ denote intuitionistic provability) if and only if the sequent $\mathcal{P} \longrightarrow G$ has a cut-free proof in which every sequent in the proof has an antecedent that is a subset of $\mathcal{D}$ and a succedent that is a member of $\mathcal{G}$. Furthermore, cut-free proofs for $\mathcal{P} \longrightarrow G$ can be searched for in a goal-directed fashion (see [14] for a more formal treatment of the relation between logic programming and goal-directed search). Since intuitionistic provability admits goal-directed theorem provers in this setting, we shall refer to the triple $\mathcal{L} = \langle \mathcal{D}, \mathcal{G}, \vdash_I \rangle$ as a logic programming language.

In presenting example programs and goals of $\mathcal{L}$, we shall use a slightly extended version of usual Prolog syntax [22]. In particular, we use the symbol `=>` for implications at the top-level of goals. Thus, we have two notations for implication: `=>` is the converse of `:-`. When denoting Horn clauses, explicit quantifiers are generally not needed, while in $\mathcal{L}$, quantifiers in both D- and G-formulas must often be made explicit. In these cases, we use the syntax `all x,y,z\` to denote universal quantification (of the three variables `x`, `y`, and `z`). We will use the following convention on naming bound variables: if the quantification occurs positively in a G-formula or negatively in a D-formula, we shall use a token with a lower case initial letter for the name of the quantified variable, otherwise we use a token with an upper case initial letter. This convention is only to aid readability: there is no logical status for the names of bound variables. When a token with an upper case initial letter is not explicitly quantified, it will be assumed to be universally quantified at the top of the formula it occurs in.

There are at least two different ways to interpret the goal $\forall x.G(x)$. The *extensional* interpretation is motivated by the semantics of universal quantification: $\forall x.G(x)$ is true of $\mathcal{P}$ if for all terms t, $G(t)$ is true of $\mathcal{P}$. (Often an additional predicate is supplied to restrict the domain of t). This interpretation of universal quantification is used often in database applications. See [6] for a formal treatment of this interpretation of universal quantification.

In this paper, we shall, however, use an *intensional* interpretation of universal quantification that is motivated by proof theory: $\forall x.G(x)$ follows from $\mathcal{P}$ if $G(c)$ follows from $\mathcal{P}$ for some constant c that does not occurs in G or $\mathcal{P}$. That is, $\forall x.G(x)$ follows if it follows generically. This interpretation of universal quantification in goals is similar to the interpretation of implications in goals used in this paper: the goal $D \supset G$ follows from program $\mathcal{P}$ if G follows for the augmented program $\mathcal{P} \cup \{D\}$.

3. Two simple examples

For a simple example, consider the familiar sterile jar problem. Assume that a jar is sterile if every germ in it is dead, that a germ in a heated jar is dead, and that a given jar has been heated. What reasoning is necessary to establish that the given jar is sterile? The intensional interpretation of the quantification will work here. Let $\mathcal{P}$ be the following program:

```
sterile(Y) :- all x\ (germ(x) => in(x,Y) => dead(x)).
dead(X)    :- heated(Y), in(X,Y), germ(X).
heated(j).
```

Consider proving the goal `?- sterile(j)`. Backchaining on the first clause above yields the goal

```
    ?- all x\ (germ(x) => in(x,j) => dead(x)).
```

Given the intensional interpretation of universal quantification, we proceed by selecting a constant, say `g`, that does not occur in $\mathcal{P}$ or in the goal. We now attempt to prove the goal

```
    ?- germ(g) => in(g,j) => dead(g).
```

This goal succeeds if the goal `dead(g)` follows from the augmented program $\mathcal{P} \cup \{$`germ(g), in(g,j)`$\}$. It is easy to see that this in fact follows by simple backchaining steps. After this goal succeeds, the two clauses `germ(g)` and `in(g,j)` are removed from the current program: the constant `g` is similarly removed (discharged).

Interpreters for $\mathcal{L}$ must use unification and free variables carefully. For example, there is no substitution for `X` such that the goal

```
?-  all y\(p(f(y)) => p(X)).
```

would succeed from the empty program. If we naively simplify this goal using the motivation above, we would first generate a new constants, say `c`, and then try to prove `p(X)` from `p(f(c))`. But this reducted problem is satisfied with the substitution of `f(c)` for `X`. Notice, however, that the result of applying this substitution to the goal above, namely

```
?-  all y\(p(f(y)) => p(f(c))).
```

does not yield a provable goal. The unsoundness arise from the fact that when `c` was selected, the future instantiations of `X` must be restricted to be terms that cannot contain the constant `c`. This restriction, which blocks the only route to a proof of the above goal, is central to most of the uses made of universals in goal in this paper.

In general, whenever a new constant is used to instantiate a universal goal, all free variables, in the goal and the program, must be restricted so that the substitution terms that will eventually instantiate them will not contain that new constant. Free variables generated by subsequent backchaining steps, however, may be instantiated with terms containing this new constant. An interpreter that restricts substitution variables for free variables as motivated above is described in the next section.

4. An interpreter for $\mathcal{L}$

In order to interprete logic programs in $\mathcal{L}$, it is necessary, in some fashion, to keep track of notions such as the "current goal," the "current program," the "current set of constants," and restrictions on free variables. Interpreters for Horn clauses only need to keep track of the first of these: there the current program and set of constants remains unchanged during a computation, and the restriction on free variables do not need to be made. In the description of an interpreter for $\mathcal{L}$ given below, a *signature* is used to denote the current set of constants, an *assignment* is used to encode the restrictions on free variables, and a *sequent* is used to connect a program to a goal.

A *signature* is a (possibly infinite) non-empty set of individual, function, and predicate constants such that there are denumerably many individual, denumerably many function, and denumerably many predicate constants of our logic that are not in the signature. The interpreter described below will need to select constants that are not already mentioned in a given signature: this last restriction on signatures makes this possible. Let Σ be a signature. A Σ-*assignment* is a finite list $\mathcal{A} = \langle t_1 \colon \Sigma_1, \ldots, t_n \colon \Sigma_n \rangle$ where $\Sigma_1 \subseteq \ldots \subseteq \Sigma_n \subseteq \Sigma$ and for $i = 1, \ldots, n$, t_i is a first-order term or atom all of whose individual, function, and predicate constants are members of Σ_i. If for some $i = 1, \ldots, n$, x occurs free in t_i then x is *assigned by* $\mathcal{A}$. If σ is a substitution, then $\sigma\langle t_1 \colon \Sigma_1, \ldots, t_n \colon \Sigma_n \rangle$ is the structure $\langle \sigma t_1 \colon \Sigma_1, \ldots, \sigma t_n \colon \Sigma_n \rangle$. If this structure is also a Σ-assignment, σ is $\mathcal{A}$-*feasible* (the value of Σ is not needed to determine $\mathcal{A}$-feasible). The expression $\mathcal{A} + \mathcal{A}'$ denotes the concatenation of the two lists $\mathcal{A}$ and $\mathcal{A}'$, and the expression $\mathcal{A} + t \colon \Sigma'$ denotes $\mathcal{A} + \langle t \colon \Sigma' \rangle$. The concatenation of two assignments is not necessarily another assignment.

The restrictions on free variables described in the previous section was given in a negative sense: a free variable is restricted to *not* be instantiated with terms containing certain constants. Σ-assignments express this restriction in an equivalent but positive fashion: if x is free in t and the pair $t \colon \Sigma'$ is a member of a Σ-assignment $\mathcal{A}$, then x can be instantiated with an term whose constants are from the set Σ'. The restriction on variables comes from the fact that only $\mathcal{A}$-feasible subsitutions will be used in the interpreter (see the BACKCHAIN transition below) and the fact that Σ' may be a proper subset of Σ.

A $\Sigma, \mathcal{A}$-sequent is a pair $\mathcal{P} \longrightarrow G$ where $G \in \mathcal{G}$, $\mathcal{P}$ is a finite subset of $\mathcal{D}$, all constants in formulas of $\mathcal{P} \cup \{G\}$ are members of Σ, and all free variables of those formulas are assigned by $\mathcal{A}$. A *state* (of the interpreter) is a triple $\langle \Sigma, \mathcal{A}, \mathcal{S} \rangle$ where Σ is a signature, $\mathcal{A}$ is a Σ-assignment, and $\mathcal{S}$ is a finite set of $\Sigma, \mathcal{A}$-sequents. These sequents specify what remains to be proved. A *success state* is a state in which the set of sequents is empty.

We assume the usual notions of substitution into first-order (quantified) formulas, first-order unification, and most general unifiers (see, for example, [22]). Simultaneous substitutions are denoted as $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$.

A simple *elaboration* function elab that maps $\mathcal{D}$ to finite subsets of $\mathcal{D}$ is defined using the equations

- $\text{elab}(A) = \{A\}$,
- $\text{elab}(G \supset A) = \{G \supset A\}$,
- $\text{elab}(D_1 \wedge D_2) = \text{elab}(D_1) \cup \text{elab}(D_2)$,
- $\text{elab}(\forall x(D_1 \wedge D_2)) = \text{elab}(\forall x.D_1) \cup \text{elab}(\forall x.D_2)$, and
- $\text{elab}(\forall x.D) = \{\forall x.D' \mid D' \in \text{elab}(D)\}$ (provided D is not conjunctive).

Elaboration simply breaks a D-formula into its conjuncts, mini-scoping outermost universal quantifiers if possible. The logical consequences of D and $\text{elab}(D)$ are the same, and a proof involving D differs in trivial ways from a proof involving $\text{elab}(D)$.

The following transition rules, indicated by $\Longrightarrow$, describe the heart of a non-deterministic interpreter. $\uplus$ denotes disjoint union.

AND: $\langle \Sigma, \mathcal{A}, \{\mathcal{P} \longrightarrow G_1 \wedge G_2\} \uplus \mathcal{S}\rangle \Longrightarrow \langle \Sigma, \mathcal{A}, \{\mathcal{P} \longrightarrow G_1, \mathcal{P} \longrightarrow G_2\} \cup \mathcal{S}\rangle$.

This transition simply translates the logical connective $\wedge$ into an AND-node in the interpreter's search space.

AUGMENT: $\langle \Sigma, \mathcal{A}, \{\mathcal{P} \longrightarrow D \supset G\} \uplus \mathcal{S}\rangle \Longrightarrow \langle \Sigma, \mathcal{A}, \{\text{elab}(D) \cup \mathcal{P} \longrightarrow G\} \cup \mathcal{S}\rangle$.

A implication in a goal is thus an instruction to augment the program with the antecedent of the implication. To simplify the presentation of backchaining below, we augment the programs clauses in $\text{elab}(D)$ instead of D.

GENERIC: $\langle \Sigma, \mathcal{A}, \{\mathcal{P} \longrightarrow \forall x.G\} \uplus \mathcal{S}\rangle \Longrightarrow \langle \Sigma \cup \{c\}, \mathcal{A}, \{\mathcal{P} \longrightarrow [x \mapsto c]G\} \cup \mathcal{S}\rangle$, provided that $c \notin \Sigma$.

A universal quantifier in a goal causes a new constant to be added to the current signature. Notice that the assignment $\mathcal{A}$ does not change; that is, the range for substitution terms for free variables does not change with this addition.

BACKCHAIN: Consider the state $\langle \Sigma, \mathcal{A}, \mathcal{S}\rangle$ where $\mathcal{S}$ is the set

$$\{\{\forall x_1 \ldots \forall x_n(G_1 \wedge \ldots \wedge G_m) \supset A\} \cup \mathcal{P} \longrightarrow A'\} \uplus \mathcal{S}'$$

for some set $\mathcal{S}'$ and for $n, m \geq 0$. Let $z_1, \ldots, z_n$ be new individual variables (that is, variables not assigned by $\mathcal{A}$) and let θ be the renaming substitution $[x_1 \mapsto z_1, \ldots, x_n \mapsto z_n]$. If θA and A' are unifiable, let σ be their most general unifier. Then the state

$$\langle \Sigma, \sigma(\mathcal{A}{+}z_1{:}\,\Sigma{+}\ldots{+}z_n{:}\,\Sigma), \sigma(\{\mathcal{P} \longrightarrow \theta G_1, \ldots, \mathcal{P} \longrightarrow \theta G_m\} \cup \mathcal{S}')\rangle$$

arises from $\langle \Sigma, \mathcal{A}, \mathcal{S}\rangle$ provided that σ is $\mathcal{A}$-feasible. If $m = 0$ then the set of sequents has diminished by one. (The application of σ

to a set of sequents, say $\mathcal{S}$, is the set of sequents resulting from applying σ to all formulas in all the sequents of $\mathcal{S}$.)

Backchaining in $\mathcal{L}$ is essentially the same as it is with Horn clauses. The main difference is that the new variables $z_1, \ldots, z_n$ must be assigned: they are allowed to be instantiated with any term involving constants in the current signature.

No transition can be applied to a success state. The following theorem is stated without proof.

Theorem. *Let G be a member of $\mathcal{G}$, $\mathcal{P}$ be a finite subset of $\mathcal{D}$, Σ a signature that contains at least the individual, function, and predicate constants occurring in G and in formulas of $\mathcal{P}$, and let $x_1, \ldots, x_n$ be a list of individual variables occurring free in G and in formulas of $\mathcal{P}$. There is a substitution σ such σG is intuitionistically derivable from $\sigma\mathcal{P}$ if and only if there is a series of transitions that carries the state $\langle \Sigma, \langle x_1 \colon \Sigma, \ldots, x_n \colon \Sigma \rangle, \{\mathcal{P} \longrightarrow G\} \rangle$ to the success state*

$$\langle \Sigma', \langle t_1 \colon \Sigma, \ldots, t_n \colon \Sigma \rangle + \mathcal{A}, \emptyset \rangle$$

such that the substitution $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ is more general than σ.

The intuitionistic logic used in this theorem is higher-order, although the higher-order aspects of that logic that are used are very weak.

We can now describe a simple, depth-first, deterministic interpreter for $\mathcal{L}$. First, we must consider the third component of a state and the antecedent of sequents as lists instead of sets. AUGMENT concatenates elaborated clauses to the front of an antecedent. When given a non-success state, the first sequent is used to determine which transition to consider. If the succedent of that sequent is an implication, apply AUGMENT; if it is a conjunction, apply AND; if it is universally quantified, apply GENERIC. The choice of constant used in GENERIC is immaterial (as long as it is not in the current signature). Finally, if the succedent is an atom, then we need to backchain. Here, we select a D-formula from the antecedent in a left-to-right order. The only backtrack points we must store are those involved with the selection of a clause: these backtrack points will be returned to following the depth-first discipline.

Notice that first-order unification does not need to be modified, although before a unifier is used in BACKCHAIN, it must be checked for $\mathcal{A}$-feasibility. This check, which provides the security used to implement data abstraction describe later, is done at runtime. Although there may be static, comile-time checks that might tell us that in certain programs feasibility of substitutions do not need to be checked, runtime checks would be necessary, in general. Also the cost of checking feasibility of substitutions is similar to the cost of doing the occur check in unification: the entire terms involved in a unifier must be transversed in order to determine that certain constants do not occur with them. It is, of course, possible to modify first-order unification so that only $\mathcal{A}$-feasible substitutions are produced. See [10, 11] for an account of how this can be accomplished. Skolem functions provide only one of several implementation techniques.

5. Local declaration of programs

A standard way to write the `reverse(L,K)` program in Prolog is to first write a tail recursive auxiliary function `rev(L,K,Acc)`. Although this second program is intended to be used only locally in the definition of `reverse`, there is no way in simple Horn clause logic or in most Prolog implementations for the scope of `rev` to be localized to just the definition of `reverse`. Making use of the universal quantification of predicates and of implications in goals, we can write a version of `reverse` where `rev` is given local scope. Consider the following D-formula.

```
reverse(L,K) :-
  all rev\(
          (all L\      (rev([],L,L)),
           all X,L,K,M\(rev([X|L],K,M) :- rev(L,K,[X|M])))
         => rev(L,K,[]))
```

(Notice that the variables `L` and `K` are bound with different scopes in this clause.) In attempting to prove the goal `reverse([1,2,3],K)` from this clause, an interpreter would first generate a new predicate symbol, say `c`, then add the Horn clauses

```
c([],L,L).
c([X|L],K,M) :- c(L,K,[X|M]).
```

to the current program, and then try to prove `c([1,2,3],K,[])`. After the answer substitution `K = [3,2,1]` is discovered, both `c` and the new clauses pertaining to `c` would be discharged.

Given this style of programming, there is another way that `reverse` can be written. One way to reverse a list, say `[a,b,c]`, is to start with the atom `rv([],[a,b,c])` and forwardchain over the clause

```
rv([X|N],M) :- rv(N,[X|M]).
```

The goal `rv([c,b,a],[])` is provable in this way. Obviously, for any list `L`, if we start with the atomic fact `rv([],L)` and forwardchain over the above clause, we can prove the atomic goal `rv(K,[])` where `L` and `K` are reverses of each other. While this is a natural approach to specifying `reverse`, it is not possible to code it directly in Horn clauses since it describes the `reverse` predicate as relating a list contained in a program and one contained in a goal. Using $\mathcal{L}$, this algorithm can be specified directly as follows.

```
reverse(L,K) :-
  all rv\ (
           (          rv([],K),
            all X,N,M\(rv([X|N],M) :- rv(N,[X|M])))
        => rv(L,[]))
```

In attempting to prove the goal `reverse([1,2,3],K)` from this clause, an interpreter will again generate a new predicate symbol, say `c`, then add the Horn clauses (where quantification is made explicit)

```
          c([],K).
all X,N,M\(c([X|N],M) :-c(N,[X|M])))
```

to the current program, and then try to prove goal `c([1,2,3],[])`. Notice here that the goal is closed while the program is open: the free variable in the program, the variable `K` in the first clause of `c`, will be instantiated to the list `[3,2,1]` by the interpreter in the process of establishing the goal `c([1,2,3],[])`. In the first clause above, `K` should not be assumed to be universally quantified: that clause is, instead, an open atomic formula.

For two more simple examples, consider how to specify goals that fail in all program contexts or that succeed only once in all program contexts. A predicate, say `fail`, will fail if there are no clauses defining it. In a dynamic setting where implications allow new clauses to be added, there is no guarantee that clauses defining `fail` are not added during some computation. The goal `all p\ p`, however, will fail in all programming contexts: when the interpreter encounters this goal, it must select a new null-ary predicate, say `c`, and then attempt to prove `c`, an attempt that must fail since `c` is new. Similarly, the goal `all p\(p => p)` will succeed exactly once in all programming context: again the interpreter will need to select a new null-ary predicate, say c, then assume `c` and then attempt to prove `c`, which will, of course, have exactly one proof in all programming contexts.

6. A mechanism for abstract data types

Universals in goals can provide a scope for constants within goal formulas. It would, of course, be useful to have a similar scoping mechanism that works over program clauses. A notion of "local" declaration for constants in a collection of program clauses is presented below.

Assume that the variable y is free in the formula D but not in the formulas G. The interpreter attempting to prove $\forall y(D \supset G)$ will then introduce a new constant for y, say k, and restrict all the current free variables so that they cannot be instantiated with terms containing k. The program code $[y \mapsto k]D$ can use the constant k to build data structures but any answer substitutions for this compound goal cannot make reference to k. It is in this sense that data abstraction can be accomodated in $\mathcal{L}$.

Before presenting some examples, it is helpful to simplify a problem of scoping. In the discussions above, the scope of y is, in a sense, only over D while we needed to use the universal quantifier $\forall y$ over the compound formula $D \supset G$ even though y is not free in G. To provide for a more natural scoping mechanism, we shall allow limited forms of existential quantification over D formulas. This example could thus be written more naturally as $(\exists y\ D) \supset G$. This use of existential quantification is also justified by the intuitionistic equivalence

$$(\exists x\ D) \supset G \quad\equiv\quad \forall x(D \supset G),$$

provided x is not free in G.

To be precise, let E be a syntactic formula variable whose range is determined by

$$E := D \mid \exists y\ E,$$

where the quantifier $\exists y$ is over individuals, functions, and predicates. The phrase "program clause" will now refer to any E formula all of whose free variables are individual variables. The interpreter would also need to make the following transition:

LOCAL: $\langle \Sigma, \mathcal{A}, \{\mathcal{P} \longrightarrow (\exists x.E) \supset G\} \uplus \mathcal{S}\rangle \Longrightarrow \langle \Sigma \cup \{c\}, \mathcal{A}, \{\mathcal{P} \longrightarrow ([x \mapsto c]E) \supset G\} \cup \mathcal{S}\rangle$, provided that $c \notin \Sigma$.

The following existentially quantified set of Horn clauses provide an implementation of the stack data type in which the constructors for stacks are not available to programs making use of this implementation.

```
exists emp, stk\(
          empty(emp),
  all S,X\( enter(X,S,stk(X,S))  ),
  all S,X\( remove(X,stk(X,S),S) )
  ).
```

Let this E-formula be denoted by the symbol `stack`. In a sense, `stack` represents a module with a local declaration. The only "exportables" constants of this module are the three predicates `empty`, `enter`, and `remove`.

A goal of the form `stack => G` is attempted by introducing two new constants that will play the role of the stack constructors, disallow the current free variables of `G` (and of the current program) to contain these constructors, and introduce three atomic clauses to implement `empty`, `enter`, and `remove`. After this point, any new free variables (introduced by subsequent backchaining steps) can be instantiated with stack objects: this is how stacks would be used in computations.

This approach to programming is, of course, very desirable since it can be used to guarantee that a client program of `stack` does not examine and manipulate stacks in any way other than those supplied by the predicates `empty`, `enter`, and `remove`. This allows different implementations of those predicates to be substituted for the module `stack`. For example, those operations could be implemented as a queue by the following code (the term `qu(L,K)` is a difference list construction):

```
exists qu\(
     all L\( empty(qu(L,L))),
  all X,L,K\( enter(X,qu(L,[X|K]),qu(L,K))  ),
  all X,L,K\( remove(X,qu([X|L],K),qu(L,K)) )
  ).
```

A search program written in $\mathcal{L}$ that uses `enter` and `remove` for storing and retrieving choice points could switch between a depth-first and breadth-first search by switching between these two implementations of those predicates.

7. Encapsulation of state

In this section, we shall make our logic language slightly higher-order in the sense that we shall allow quantification over propositional variables in D-formulas and permit predicate constants to appear within terms. Operationally speaking, we are making this extension to allow goal formulas to be passed around as arguments and to be dynamically called. Various higher-order extensions to logic programming have been analyzed in the papers [12, 14, 16]. The extension mentioned above is part of the much more general theory of *higher-order hereditary Harrop formulas* described in [14]. Although there is not sufficient space here to present details, it suffices to say that when propositional variables are not permitted as the head of definite clauses and when there are no logical constants embedded inside the terms of the logic (both cases are true of the examples below), then the straightforward operational meaning of these extended definite clauses can be given a proof theoretic semantics.

The following is an implementation of a switch data type where a switch's value has a scope. Consider the following program clauses:

```
exists sw\(
                 sw(off),
        all G\( set_on(G)  :- sw(on)  => G ),
        all G\( set_off(G) :- sw(off) => G ),
        all V\( status(V)  :- sw(V) )
   ).
```

The value for this switch is stored as the argument for the local, one-place predicate `sw`. The switch is initially set off by the first line. The predicates `set_on` and `set_off` take a goal formula as their argument (hence, the need for the higher-order extensions), set the switch either on or off by extending the program, and then call their arguments. Propositional variables allow a kind of "continuation passing" style of programming.

Notice that as a series of `set_on` and `set_off` predicates are called, there is an accumulation of all the previous settings of the switch. In a sense, when the switch gets set, it becomes more non-deterministic. In order to get the more deterministic and coventional notion of a switch we must consider various schemes for reducing non-determinism. There seems to be two natural choices for doing this. First, implication could be interpreted as redefining instead of augmenting. Many of the previous examples still have interesting meaning under such a reinterpretation of implication. The other choice, used here, is to provide the deterministic version of the intepreter with such control primitives as the "deterministic" declaration or cut (!).

As it is implemented above, the `status` predicate is the only way the value of the current switch can be determined. If the goal `?- status(U)` is called, `U` will be bound to the most recent setting of the switch. Notice, however, that the call `?- status(on)` succeeds if the switch had been set on at some point. If `status` were reimplemented using cut as

```
     all U,V\( status(V)  :- sw(U), !, U = V )
```

only the last value of the switch could ever be retrieved (by the deterministic interpreter).

Notice that, in general, the entire history of how this switch is set must be maintained since completing a goal such as `set_on(G)` requires a previous switch value to be reinstated. If the deterministic version of `status` is used and it is known that the goals called as continuations in `set_on` and `set_off` never fail, then previous settings of the switch are not needed. In this case, `set_on` and `set_off` could be implemented using a side-effect to change the argument of the local predicate `sw`.

For a final example, consider the following simple exercise in using a similar form of encapsulation.

```
make_account(Acc,Amt,G) :- all reg\ (
  (  reg(Amt),
     all Inc, H,  Val, Tmp\(
        add_money(Acc,Inc,H) :-
           reg(Val), Tmp is (Val + Inc), reg(Tmp) => H),
     all Dec, H, Val, Tmp\(
        wd_money(Acc,Dec,H) :-
           reg(Val), Tmp is (Val - Dec), reg(Tmp) => H ),
     all H, Val\(
        print_amt(Acc,H) :-
           reg(Val), write(Val), nl, H)
   => G).
```

The goal `make_account(john,100,G)` would call the goal `G` in an environment where there is an “account” named `john` that is initialized with the amount 100. This account is stored as a local predicate, which stores the balance (or state) of the account, and three “methods” for adding to, subtracting from, and printing that account’s balance. The continuation `G` is given access to the three predicates `add_money`, `wd_money`, and `print_amt`. If `G` itself calls `make_account`, a new local predicate and three new “methods” are created to implement the new account.

The following is a very simple interpreter for treating the named accounts used as objects. In this example, the only continuation called is the predicate `transact`.

```
transact :-
   write(">>- "), read(Entry), do(Entry).
do(mk_acc(Name,Amt)) :- make_account(Name,Amt,transact).
do(add(Name,Amt)) :- ad_money(Name,Amt,transact).
do(wd(Name,Amt)) :- wd_money(Name,Amt,transact).
do(print(Name)) :- print_amt(Name,transact).
do(quit).
```

The following is a simple interaction with this transaction program.

```
?- transact.
>>- mk_acc(john,10).
>>- mk_acc(mary,20).
>>- add(john,5).
>>- print(john).
15
>>- wd(mary,10).
>>- print(mary).
10
>>- quit.
?-
```

Again, if the continuation `transact` never fails (that is, the user only types in correct information), then only the most recent state of an account is examined.

Acknowledgements. I would like to thank John Hannan, Frank Pfenning, and several of the attendees of the GULP Advanced School on Foundations of Logic Programming, September 1988 for their helpful comments and criticism of the work described in this paper. This work is supported in part by grants ONR N00014-88-K-0633, NSF CCR-87-05596, and DARPA N00014-85-K-0018.

8. References

[1] C. Elliott and F. Pfenning, eLP, a Common Lisp implementation of λProlog, January 1989.

[2] A. Felty and D. Miller, Specifying Theorem Provers in a Higher-Order Logic Programming Language, Proceedings of the Ninth International Conference on Automated Deduction, Argonne, IL, 23 – 26 May 1988.

[3] D. Gabbay and U. Reyle, N-Prolog: An Extension to Prolog with Hypothetical Implications. I, Journal of Logic Programming 1, 1984, 319 – 355.

[4] L. Giordano, A. Martelli, and G. Rossi, Local Definitions with Static Scope Rules in Logic Programming, Proceedings of the FGCS International Conference, Tokyo, 1988, pp. 389-396.

[5] L. Hallnäs and P. Schroeder-Heister, A Proof-Theoretic Approach to Logic Programming. I: Generalized Horn Clauses (unpublished).

[6] J. Lloyd and R. Topor, Making Prolog More Expressive, Journal of Logic Programming 1(3), October 1984, 225 – 240.

[7] L. McCarty, Clausal Intuitionistic Logic I. Fixed Point Semantics, Journal of Logic Programming 5(1), March 1988, 1 – 31.

[8] L. McCarty, Clausal Intuitionistic Logic II. Tableau Proof Procedure, Journal of Logic Programming 5, 93 – 132, 1988.

[9] D. Miller, A Logical Analysis of Modules in Logic Programming, Journal of Logic Programming 6 (1989), 79 – 108.

[10] D. Miller, Solutions to λ-term equations under a mixed prefix (submitted, January 1989).

[11] D. Miller, Unification under a mixed prefix (unpublished, December 1988).

[12] D. Miller and G. Nadathur, Higher-order Logic Programming, Proceedings of the Third International Logic Programming Conference, London, June 1986, 448 – 462.

[13] D. Miller and G. Nadathur, LP2.7, a C-Prolog and Quintus Prolog implementation of λProlog (July 1988).

[14] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, Uniform Proofs as a Foundations for Logic Programming, Annals of Pure and Applied Logic (to appear).

[15] L. Monteiro and A. Porto, Contextual Logic Programming, Proceedings of the Sixth International Logic Programming Conference, Lisbon Portugal, June 1989.

[16] G. Nadathur, A Higher-Order Logic as the Basis for Logic Programming, Ph.D. dissertation, University of Pennsylvania, May 1987.

[17] G. Nadathur and D. Miller, An Overview of λProlog, Fifth International Conference on Logic Programming, MIT Press, 1988.

[18] L. Pauslon, The Foundation of a Generic Theorem Prover, Journal of Automated Reasoning (to appear).

[19] F. Pfenning, Partial Polymorphic Type Inference and Higher-Order Unification, Proceedings of the 1988 ACM Conference on

Lisp and Functional Programming.

[20] D. Sannella and L. Wallen, A Calculus for the Construction of Modular Prolog Programs, Proceedings of the 1987 Symposium on Logic Programming, San Francisco, 1987.

[21] G. Smolka, TEL (Version 0.9), Report and Unser Manula. SEKI Report SR-87-11, FB Informatik, Universität Kaiserslautern, W. Germany, 1987.

[22] L. Sterling and E. Shapiro, *The art of Prolog: advanced programming techniques*, MIT Press, Cambridge MA, 1986.