PROLOG CONTROL IN SIX SLIDES

Nils M Holm, July 2019 (with corrections, August 2019)

1. Introduction

PROLOG [Col75] [Kow74] is a language for finding values that satisfy logical statements by matching them against clauses contained in a database. A PROLOG program itself consists of a set of "clauses", like

```
mortal(X) :- man(X).
```

meaning that "X is mortal, if X is a man". The "term" (or "goal") *mortal*(X) is a conclusion and *man*(X) is a premise that has to be satisfied for the conclusion to be true. There can be any number of goals in the "body" (after the ":–") of a clause:

```
likes(john, X) :- tasty(X),
                  healthy(X).
```

This one means that John likes things that are tasty *and* healthy. Capitalized symbols are variables, lower-case symbols are "atoms" (constants). A "compound term", like `tasty(X)`, consists of a "functor" (`tasty`) followed by a positive number of comma-separated "arguments" in parentheses. Clauses without any premises are called "unit clauses" or "facts":

```
tasty(bananas).
tasty(bread).
tasty(chocolate).
healthy(bananas).
healthy(porridge).
healthy(bread).
```

A clause starting with the symbol "?–" is called a "query". It is used to ask questions:

```
?- likes(john, chocolate).
no
?- likes(john, bananas).
yes
```

The true power of PROLOG is revealed when there are multiple answers that satisfy a query. The following facts represent the edges of the acyclic graph in figure 1.

```
edge(a,b).  edge(b,d).  edge(d,h).
edge(a,f).  edge(c,d).  edge(h,e).
edge(a,g).  edge(c,e).  edge(h,f).
edge(b,c).  edge(g,h).
```

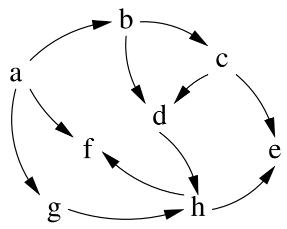

figure 1

The following program can then be used to find paths inside of the graph. It consists of two clauses. The first clause states that "there is a path [A, B] from A to B, if there is an edge from A to B":

```
path(A,B,[A,B]) :- edge(A,B).
```

The term [A, B] denotes a list containing the elements A and B. The second clause states that "[$A|CB$] is a path from A to B, if there is an edge from A to some node C and a path CB from that node C to B":

```
path(A,B,[A|CB]) :- edge(A,C),
                    path(C,B,CB).
```

The term [$A|CB$] indicates a list whose head (first element) is A and whose tail (list containing the rest of its elements) is CB. The following query finds all paths P from node a to node f:

```
?- path(a,f,P).
P = [a,f]
P = [a,b,c,d,h,f]
P = [a,b,d,h,f]
P = [a,g,h,f]
```

The query

```
?- path(A, B, P).
```

would find *all* paths P from any node A to any node B in the graph. The output is not shown here, because it is rather lengthy.

The PROLOG system finds all possible solutions satisfying a query by systematically trying them and backtracking when a partial solutions leads to a contradiction.

2. Backtracking

From an implementor's point of view, PROLOG proves the conjunction of goals $g_1, \cdots$ in the disjunction of rules $r_1, \cdots$, which is often thought of as an AND/OR tree structure [KK71] as depicted in figure 2. g denotes a goal and r a rule that is used in an attempt to prove the goal. A solution is a path through the tree from the root to a leaf.

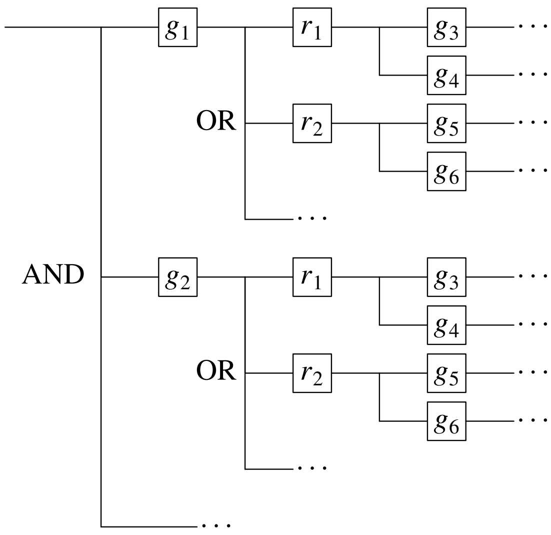

figure 2

Proving a goal g using a rule r removes g from the set of goals to be proven, but also adds the goals in the body of r to the set. Only facts add no new goals. They form the leaves of the tree. In order to prove a set of goals, all rules are tried on every goal until either the set of goals to be proven is empty or there are no rules left to try. This is depicted in figure 3.

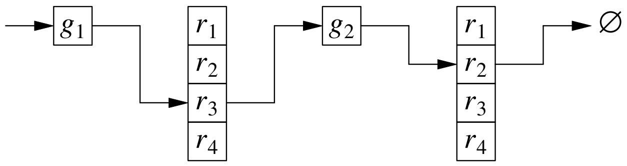

figure 3

In the figure a solution is found by applying rule r_3 to g_1 and r_2 to g_2. After g_2 there are no more goals to prove. However, additional solutions might be found by proving g_2 using r_3 or r_4 or by proving g_1 using r_4, which would offer a completely new set of options, because g_2 could now be proven with each rule again.

The process that finds all solutions systematically is called "backtracking". It finds the first rule r satisfying a goal g and then proves any remaining goals in the "context" of proving g using r. The context of a proof is the set of bindings from variables to values created during unification. When unification of g and r fails, the next rule is tried and when no more rules exist, PROLOG backtracks to the previous goal, returning to a prior context and trying a different rule.

3. Unification

Unification [Rob65] is a process that attempts to match two terms, like the head (conclusion) of a clause and a goal. The process is described in detail in the literature, so it shall only be outlined at this point.

Unification "succeeds", if two terms u and v can be unified. Otherwise it fails. The terms u and v can be unified, if

– u and v are the same atom
– u and v are variables bound to the same value
– either u or v is an unbound variable
– both u and v are compound terms whose elements can be unified

When an unbound variable v is unified with some term u, then u will become the value of v; v will be "bound to" u. If both terms are variables, they will "co-refer" after the unification, i.e. they will always bind to the same value.

A simple unification algorithm is given in the appendix.

4. Recursive Control

"Control" [Kow79] is the part of a PROLOG system that controls the process of finding solutions that satisfy a set of goals. It is the part of the system that decides when to unify, when to backtrack, when to halt, and where to backtrack to.

PROLOG control can easily be expressed as a set of two recursive functions [Kah83] [BDF93], *prove* and *try*, which is given here in R4RS Scheme [CR91]:

```
(define (try g r e n)
  (if (null? r)
      #f
      (let* ((a  (copy (car r)
                       (list n)))
             (ne (unify (car g)
                        (car a)
                        e)))
        (if ne
            (prove3 (append
                       (cdr a)
                       (cdr g))
                     ne
                     (+ 1 n)))
        (try g
             (cdr r)
             e
             n))))

(define (prove3 g e n)
  (cond ((null? g)
          (print-frame e))
        (else
          (try g db e n))))
```

The *prove* function attempts to prove the goals in the list *g* in the environment (context) *e* given the rules in the list (database) *db*. The variable *n* is used to copy the structure of a rule in order to make its variable names unique, i.e. it turns

`f(X) :- g(X), h(X).`

into

f(X$_n$) :- g(X$_n$), h(X$_n$).

The algorithm of *try* is basically this:

– unify g_1 with a copy of r_i, giving a new context e'
– if unification succeeds, prove the remaining goals and the body of r_i in e'
– try the next rule r_{i+1} in *e*

The algorithm of *prove* is simple: when the list of goals is empty, print all ‘‘instantiations’’ (bindings) of the variables in the query and else try to prove *g* in every rule contained in the database *db*.

Prove calls *try* to prove a sequence of goals using any rule and *try* calls *prove* to add the body of a rule to the set of goals to be proven. *Try* always tries *all* rules, thereby searching the entire solution space systematically.

When no goals remain to be proved, the instantiations of the query variables form a solution satisfying all goals of the query. The try/prove algorithms halts when it runs out of rules.

The algorithm is straight-forward, but it relies heavily on recursion, which causes a few problems:

(1) activation frames are allocated on the stack, which may overflow while running a proof
(2) flow of control is implied, so this model does not map well to low-level language implementations
(3) extensions that change the flow of control (like the ‘‘cut’’) would require catch/throw, continuations, or a simlar mechanism

There are other, more explicit models for implementing PROLOG, most prominently the Warren Abstract Machine (WAM) [War83], which is notoriously hard to implement [Ait91], as well as several simplified models of the WAM, which are still quite complex. For instance, the implementation in [BDF93] still has a size of 670 lines.

5. Control in Six Slides

The control algorithm presented here is explicit, simple, and short. Its basic mode of operation can be illustrated in six small panels, it keeps its state in a stack structure that is under control of the program, so it maps well to low-level languages, and it makes control explicit, so that extensions like the cut can be integrated easily.

5.1. Slide 1: Registers

L	*link*, (L, G, R, E, N)
G	*goals*, $\{g_1, \cdots\}$
R	*rules*, $\{r_1, \cdots\}$
E	*environment*
N	*time stamp*

slide 1: registers

The model presented here includes the same registers as the recursive control algorithm: **G** is a list of goals to prove, **R** is a list of rules available for proving goals, **E** is an environment containing bindings from variables to values, and **N** is a time stamp that is used to make variables in rules unique.

There is one additional register, **L**, which serves as a *link* between contexts. A “context” is a structure containing all of the above registers, including **L** itself. Hence it captures the entire state of a computation at a given point. Because it contains its previous (outer) context as one of its components, it implements the call stack of the interpreter.

Initially, $\mathbf{L} = \varnothing$, **G** contains a list of goals, **R** contains a list of rules (including facts), $\mathbf{E} = \varnothing$, and $\mathbf{N} = 0$. $\varnothing$ denotes the empty list.

5.2. Slide 2: Success

When unification of the first goal in **G** and the first rule in **R** succeeds, the new state of the system is constructed as follows:

L captures the current context. **G** is set to the union of the body of the current rule and the remaining goals, i.e. the first goal is removed from **G** and the body of the current rule is added.

R is set to the entirety of rules known to the interpreter (the database *DB*), so that all rules will be tried in subsequent proof attempts. **E** will be extended with the bindings created during unification, and **N** is simply moved forward.

Slide 2, like all other slides, shows the original state of the interpreter in the left column and the new state in the right column.

$$\begin{array}{lcl} L & & (L, G, R, E, N) \\ G & & body(first(R)) \ \cup \ rest(G) \\ R & \xrightarrow{Succeed} & DB \\ E & & E' \\ N & & N+1 \end{array}$$

slide 2: success

5.3. Slide 3: Refutation

When $\mathbf{G} = \varnothing$ (the set of goals is empty), a solution to a query has been found. The instantiated query variables will be printed. After that the state saved in the **L** register will be restored and the first rule will be removed from the **R** register. This second step implements backtracking. It attempts to prove the goals L_G of the previous context using alternative rules.

$$\begin{array}{lcl} L & & L_L \\ \varnothing & & L_G \\ R & \xrightarrow[Print]{} & rest(L_R) \\ E & & L_E \\ N & & L_N \end{array}$$

slide 3: refutation

5.4. Slide 4: Failure

When unfication of the first goal in **G** and the first rule in **R** fails, then the rest of the rules is tried by removing the first rule from **R**. All other registers remain unchanged. The rest of **R** may be empty, leading to inconsistency (slide 5).

$$\begin{array}{lcl} L & & L \\ G & & G \\ R & \xrightarrow{Fail} & rest(R) \\ E & & E \\ N & & N \end{array}$$

slide 4: failure

5.5. Slide 5: Inconsistency

When $\mathbf{R} = \varnothing$ (the set of rules is empty), no consistent answer to the query could be found in the current context. The previous context is restored and the first rule (which lead to the current context) is removed from that context. This is the same backtracking step as in slide 3.

$$\begin{array}{lcl} L & & L_L \\ G & & L_G \\ \varnothing & \xrightarrow{} & rest(L_R) \\ E & & L_E \\ N & & L_N \end{array}$$

slide 5: inconsistency

5.6. Slide 6: Termination

When **R** = ∅ *and* **L** = ∅, i.e. there are no rules left *and* there is no context to backtrack to, so the computation is finished.

$$\begin{tikzcd}
\varnothing \\
G \\
\varnothing \arrow[r] & \mathit{halt} \\
E \\
N
\end{tikzcd}$$

slide 6: termination

6. Implementation

The **link** structure is implemented as a list for simplicity's sake. Vectors, records, or even a stack of context frames may be used instead.

```
(define link list)
(define L_l car)
(define L_g cadr)
(define L_r caddr)
(define L_e cadddr)
(define (L_n x) (car (cddddr x)))
```

The **back** procedure implements backtracking as shown in slides 3, 4, and 5. It also handles the simple case where goals are left to prove *and* rules are left to try (slide 4) by just advancing to the next rule.

```
(define (back5 l g r e n)
  (if (and (pair? g)
           (pair? r))
      (prove5 l g (cdr r) e n)
      (prove5 (L_l l)
              (L_g l)
              (cdr (L_r l))
              (L_e l)
              (L_n l))))
```

The **prove** procedure first handles the cases

G = ∅
R = ∅ and **L** ≠ ∅
R = ∅ and **L** = ∅

as specified in slides 3, 5, and 6.

Its default case attempts to unify the head of the first rule with the first goal. Depending on the result of the unification, it implements slides 2 and 4.

```
(define (prove5 l g r e n)
  (cond
    ((null? g)
      (print-frame e)
      (back5 l g r e n))
    ((null? r)
      (if (null? l)
          #t
          (back5 l g r e n)))
    (else
      (let* ((a  (copy (car r) n))
             (e* (unify (car a)
                        (car g)
                        e)))
        (if e*
            (prove5
              (link l g r e n)
              (append (cdr a)
                      (cdr g))
              db
              e*
              (+ 1 n))
            (back5 l g r e n))))))
```

The **prove** procedure is tail-recursive and **prove** and **back** are mutually tail-recursive. Hence the implementation does not make use of the internal stack of the implementation language at all. The **prove** procedure is merely a loop that modifies the state of the **L**, **G**, **R**, **E**, **N** registers and **back** is a convenience procedure that simplifies the code of **prove**. It could be inlined in **prove** without any further consequences.

7. Control with Cut in Nine Slides

The "cut" operator "!" is a goal that always succeeds, but when backtracking over it, all goals to its left will be ignored. For instance,

```
v(a). v(b).
?- v(X), v(Y).
```

will produce the following answers: $(X = a, Y = a)$, $(X = a, Y = b)$, $(X = b, Y = a)$, and $(X = b, Y = b)$. However,

```
?- v(X), !, v(Y).
```

will only produce the first two, because the process will never backtrack to $v(X)$.

In order to add the cut operator to the interpreter, an additional register, **C** ("cut point"), is introduced. Like the **L** register, it points to a previous context, but not necessarily the immediate outer context. It points to the most recent context where the head of a rule could be successfully unified with a goal. See figure 4. The **C** register is ∅ initially.

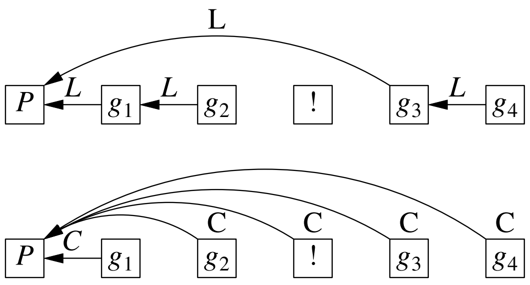

figure 4

In the figure, the link **L** of each goal context g_i points to the previous goal, except for the link of g_3, which points to the cut point P, because it is preceded by a cut. The **C** register of each context in the list of goals also points to P.

Slide 1a introduces the **C** register, which points a structure that is equal to that of the **L** register.

L	*link*, (L, G, R, E, N, C)
G	*goals*, $\{g_1, \cdots\}$
R	*rules*, $\{r_1, \cdots\}$
E	*environment*
N	*time stamp*
C	*cut point*, $(L, G, \{\varnothing\}, E, N, C)$

slide 1a: registers with cut

When a goal matches a rule, a new cut point is set up by copying the **L** register to the **C** register and setting $\mathbf{C}_R$ to $\{\varnothing\}$ (a rule set with a single empty rule). See slide 2a. The empty rule will never be matched and skipped when backtracking.

Slide 2a also injects an internal goal of the form $r!X$ into the set of goals to be proven, right after the body of *first*(R). This goal is proven when the body of *first*(R) succeeds. It will then also succeed and reset the cut point to X. When *first*(R) does not succeed, **C** will be reset by backtracking.

$$\begin{array}{lcl}
L & & (L, G, R, E, N, C) \\
G & & body(first(R)) \cup \{r!L\} \\
 & & \quad \cup \ rest(G) \\
R & \xrightarrow{Succeed} & DB \\
E & & E' \\
N & & N+1 \\
C & & \{L_L,\ L_G,\ \{\varnothing\}, L_E, L_N, L_C\}
\end{array}$$

slide 2a: success with cut

The slides 3a through 6a correspond to slides 3–6, but they also carry along the **C** register.

$$\begin{array}{lcl}
L & & L_L \\
\varnothing & & L_G \\
R & & rest(L_R) \\
E & \xrightarrow[Print]{} & L_E \\
N & & L_N \\
C & & L_C
\end{array}$$

slide 2a: refutation with cut

$$\begin{array}{lcl}
L & & L \\
G & & G \\
R & \xrightarrow{Fail} & rest(R) \\
E & & E \\
N & & N \\
C & & C
\end{array}$$

slide 4a: failure with cut

$$\begin{array}{lcl}
L & & L_L \\
G & & L_G \\
\varnothing & \longrightarrow & rest(L_R) \\
E & & L_E \\
N & & L_N \\
C & & L_C
\end{array}$$

slide 5a: inconsistency with cut

$$\begin{array}{lcl} \varnothing & & \\ G & & \\ \varnothing & \longrightarrow & \mathit{halt} \\ E & & \\ N & & \\ C & & \end{array}$$

slide 6a: termination with cut

7.1. Slide 7: Cut

When a cut operator (**!**) is encountered in **G**, the operator is removed from **G** and the cut context is copied from the **C** register to the **L** register of the following goal g. When backtracking is thereafter initiated in the context of g, control will not be transferred to the previous goal, but to the backtracking point P, as shown in figure 4.

$$\begin{array}{lcl} L & & C \\ [!, g, \cdots] & & [g, \cdots] \\ R & \longrightarrow & R \\ E & & E \\ N & & N \\ C & & C \end{array}$$

slide 7: cut

7.2. Slide 8: Reset Cut Point

When a “procedure” (a collection of clauses with a common functor) is exited, the **C** register will be reset to the cut point of the calling procedure. This is done by the internal goal $r!X$, which copies its argument X to the **C** register.

$$\begin{array}{lcl} L & & L \\ [\mathrm{r}!X, g, \cdots] & & [g, \cdots] \\ R & \longrightarrow & R \\ E & & E \\ N & & N \\ C & & X \end{array}$$

slide 8: reset cut point

7.3. Slide 9: Top-Level Cut

The following definition is a common version of the **member** predicate that will succeed only once with the first matching member:

```
member1(X, [X|T]) :- !.
member1(X, [Y|T]) :- member1(X, T).
```

For instance:

```
?- member1(X, [a,b,c]).
X = a
```

When this query is submitted directly the interpreter, then it will reach a configuration where there are no more goals to prove and there is no context to return to after backtracking over the cut, but there might still be rules left to try.

This cannot happen in the model without cut, so this special case has to be handled in an additional slide.

$$\begin{array}{lcl} \varnothing & & \\ \varnothing & & \\ R & \longrightarrow & \mathit{halt} \\ E & & \\ N & & \\ C & & \end{array}$$

slide 9: top-level cut

8. Implementation with Cut

An accessor for the **C** register is added. Accessors for the other registers remain the same as in the previous implementation.

```
(define (L_c x) (cadr (cddddr x)))
```

The **clear_r** procedure sets the **R** register of a cut point (**link** structure) to (∅). This can be done destructively, because cut points will never get “uncut”.

```
(define (clear_r x)
  (set-car! (cddr x) '(())))
```

Slide 9 is implemented in the **back** procedure. When **G** = ∅ and **L** = ∅, the procedure simply returns and, because it is mutually tail-recursive with **prove**, this means that the entire process will terminate.

```
(define (back6 l g r e n c)
  (cond
    ((and (pair? g)
          (pair? r))
      (prove6 l g (cdr r) e n c))
    ((pair? l)
      (prove6 (L_l l)
              (L_g l)
              (cdr (L_r l))
              (L_e l)
              (L_n l)
              (L_c l)))))
```

Some minor changes in **prove** implement slides 2a, 7, and 8: (2a) When **prove** recurses after successful unification it copies **L** to **C**. (7,8) Two new cases are added to handle cut (!) and reset-cut (r!) goals.

The additional cases have to be placed before the one that tests **R** = ∅, because a cut may still appear while **R** = ∅.

```
(define (prove6 l g r e n c)
  (cond
    ((null? g)
      (print-frame e)
      (back6 l g r e n c))
    ((eq? '! (car g))
      (clear_r c)
      (prove6 c (cdr g) r e n c))
    ((eq? 'r! (car g))
      (prove6 l (cddr g) r e n
              (cadr g)))
    ((null? r)
      (if (null? l)
          #t
          (back6 l g r e n c)))
    (else
      (let* ((a  (copy (car r) n))
             (e* (unify (car a)
                        (car g)
                        e)))
        (if e*
            (prove6
              (link l g r e n c)
              (append (cdr a)
                      `(r! ,l)
                      (cdr g))
              db
              e*
              (+ 1 n)
              l)
            (back6 l g r e n c))))))
```

9. Conclusion and Perspective

The implementation method presented here is simple, extensible, and suitable for many modifcations to improve performance. For instance, the **link** structure could be placed on a stack, with one of its components per stack slot, thereby eliminating all consing related to the forward and backward process. The binding method is (mostly) independent from the model, so more efficient methods can be chosen.

In an experimental implementation clause indexing and shallow binding have been successfully combined with the 9-slide algorithm. Last-call optimization should be possible, but exploring this shall be postponed to a later point.

10. Appendix: Source Code

The proof of concept code in the appendix represents clauses as Scheme lists, atoms as symbols, and variables as lists of the form

```
(? name)
```

Unit clauses are singleton lists and rules are lists with multiple elements, where the first one is the head and the rest forms the body of the rule. For instance,

```
mortal(X) :- man(X).
```

would be encoded as

```
((mortal (? x)) (man (? x)))
```

Deep-binding of variables to values is done via association lists.

10.1. Unification

```
(define empty '((bottom)))

(define var '?)
(define name cadr)
(define time cddr)

(define (var? x)
  (and (pair? x)
       (eq? var (car x))))

(define (lookup v e)
  (let ((id (name v))
        (t  (time v)))
    (let loop ((e e))
       (cond
         ((not (pair? (caar e)))
           #f)
         ((and
            (eq? id (name (caar e)))
            (eqv? t (time (caar e))))
           (car e))
         (else
           (loop (cdr e)))))))

(define (value x e)
  (if (var? x)
      (let ((v (lookup x e)))
        (if v
            (value (cadr v) e)
            x))
      x))

(define (copy x n)
  (cond
    ((not (pair? x)) x)
    ((var? x) (append x n))
    (else
      (cons (copy (car x) n)
            (copy (cdr x) n)))))

(define (bind x y e)
  (cons (list x y) e))

(define (unify x y e)
  (let ((x (value x e))
        (y (value y e)))
    (cond
      ((eq? x y) e)
      ((var? x) (bind x y e))
      ((var? y) (bind y x e))
      ((or (not (pair? x))
           (not (pair? y))) #f)
      (else
        (let ((e* (unify (car x)
                         (car y)
                         e)))
          (and e* (unify (cdr x)
                         (cdr y)
                         e*)))))))
```

10.2. Printing Frames

```
(define (resolve x e)
  (cond ((not (pair? x)) x)
        ((var? x)
          (let ((v (value x e)))
            (if (var? v)
                v
                (resolve v e))))
        (else
          (cons
            (resolve (car x) e)
            (resolve (cdr x) e)))))

(define (print-frame e)
  (newline)
  (let loop ((ee e))
    (cond
      ((pair? (cdr ee))
        (cond
          ((null? (time
                    (caar ee)))
            (display (cadaar ee))
            (display " = ")
            (display
              (resolve (caar ee)
                       e))
            (newline)))
        (loop (cdr ee))))))
```

10.3. Example Programs

```
;; Graph example from section 1

(define db
  '(((edge a b))
    ((edge a f))
    ((edge a g))
    ((edge b c))
    ((edge b d))
    ((edge c d))
    ((edge c e))
    ((edge g h))
    ((edge d h))
    ((edge h e))
    ((edge h f))
    ((path (? A)
           (? B)
           ((? A) (? B)))
     (edge (? A) (? B)))

    ((path (? A)
           (? B)
           ((? A) . (? CB)))
     (edge (? A) (? C))
     (path (? C) (? B) (? CB)))))

(define goals '((path a f (? P))))

; recursive PROVE
(prove3 goals empty 1)

; 6-slide PROVE
(prove5 '() goals db empty 1)

;; Negation as failure

(define db
  '(((some foo))
    ((some bar))
    ((some baz))

    ((eq (? X) (? X)))

    ((neq (? X) (? Y))
     (eq (? X) (? Y)) ! fail)

    ((neq (? X) (? Y)))))

(define goals '((some (? X))
                (some (? Y))
                (neq (? X) (? Y))))

; 9-slide PROVE
(prove6 '() goals db empty 1 '())
```

11. Corrections

Corrections were made to slides 2a and 7 in order to support cuts in recursive clauses. Slide 8 was added to simplify cuts (and the original slide 8 became slide 9).

12. References

[Ait91]
Hassan Ait-Kaci; “Warren’s Abstract Machine: A Tutorial Reconstruction”; MIT Press, 1991.

[BDF93]
Patrice Boizumault, Ara M. Djamboulian, Jamal Fattouh; “The Implementation of Prolog”; Princeton University Press, 1993

[Col75]
Alain Colmerauer; “Les Grammaires de Metamorphose”; Groupe d’Intelligence Artificielle, Marseille-Luminy Universite, Nov 1975; Appears as “Metamorphosis Grammars” in "Natural Language Communication with Computers", Springer 1978.

[CR91]
William Clinger, Jonathan Rees (Editors); “Revised4 Report on the Algorithmic Language Scheme”; ACM SIGPLAN Lisp Pointers, Volume IV Issue 3, July, 1991, Pages 1-55

[Kah83]
Ken Kahn; “A Pure Prolog Written In Pure Lisp”; SAIL AIList Digest V1, #47, 1983

[Kow74]
Robert Kowalski; “Logic for Problem Solving”; DCL Memo 75, Department of AI, University of Edinburgh, Mar 74.

[Kow79]
Robert Kowalski; “Algorithm = Logic + Control"; Communications of the ACM (CACM), Volume 22 Issue 7, July 1979, Pages 424-436

[KK71]
Robert Kowalski, Donald Kuehner; “Linear Resolution with Selection Function”; Artificial Intelligence 2(3):227-260, December 1971

[Rob65]
John A. Robinson; “A Machine-Oriented Logic Based on the Resolution Principle”; Journal of the ACM (JACM), Volume 12 Issue 1, Jan. 1965, Pages 23-41

[War83]
David H. D. Warren; “An Abstract Prolog Instruction Set”; Technical Note No 309, SRI International, Menlo Park, CA, 1983.