THE OVERDIRECTOR

by

Peter Bruell

August 1979 ATP-49

This work was supported in part by NSF Grant MCS77-20701 at The University of Texas at Austin.

Contents

1.0 INTRODUCTION

The OVERDIRECTOR conceptually isolates domain dependence of the search conducted by the program described in [1]. Its mechanical aspects along with its interface with IMPLY are detailed in this document. The document also contains brief descriptions of some semantic, domain dependent functions (e.g. PRIORITY) which hint at the way in which semantic knowledge is called upon. Before continuing it would be helpful to visualize the system as depicted in Figure 1.

```
        -----------------------
        |                     |
        |                     |
        |    OVERDIRECTOR     |
        |                     |
        |                     |
        -----------------------
                   |
                   |
                   V
                AGENDA
                   ^
                   |
                   |
           -----------------
           |               |
           |     IMPLY     |
           |               |
           -----------------
```

Figure 1.

When the program is running, control passes back and forth between the two components, and they communicate with one another via the AGENDA.

Section three of this document is most easily studied by first referring to figure 2. The first assignment box shows the basic task of the functions CYCLE and CYCLE*, while the big loop corresponds to the body of the function OVERDIRECTOR itself. Since the listing of functions is in alphabetical order, the suggested approach is to turn first to the description of OVERDIRECTOR and to proceed outwards from there.

```
                         |
                         |
                         |
                         |
                         V
     ----------------------------------------------------------
     |                                                        |
     |     H <= Hypothesis of theorem                         |
     |     C <= Conclusion of theorem                         |
     |     AGENDA <= NIL                                      |
     |     Generate-Tasks((Quit C T), (0 NIL NIL T))          |
     |                                                        |
     ----------------------------------------------------------
                           |
                           |
    |--------------------->|
    |                      |
    |                      V
    |          -----------------------
    |         (                       )   T
    |         (     AGENDA empty?     ) ------->  No proof found
    |         (                       )
    |          -----------------------
    |                    |
    |                    |F
    |                    |
    |                    |
    |                    V
    |       --------------------------------------
    |       |                                    |
    |       |  J <= Select-Task(T)               |
    |       |  Apply-Method(Caar(J), Cdr(J), T)  |
    |       |                                    |
    |       --------------------------------------
    |                        |
    |                        | New tasks are added to the AGENDA as
    |                        | a result of the call to Apply-Method
    |                        |
    |                        V
    |               --------------------------
    |    F         (                          )
    -------        (   TRUE-task on AGENDA?   )
                   (                          )
                    --------------------------
                           |
                           |T
                           |
                           |
                           V
                      Proof Found
```

Figure 2.

2.0 TYPE DECLARATIONS

For the purposes of documentation it will be convenient to treat LISP code as if it were written in a more formal algebraic language which admits data types. Since the documentation is not executable code, these data types will be specified informally.

```
Ans = Quit or Answer or NIL

Answer = (Sigma DB)

BC-Lemmas = "a list of names of theorems to be used for backward
             chaining" (see chapter 4 of [1])

Conclusion = "a conclusion"

DB = "a data base"

Direction = (Formula Formula) or NIL

Disjunctive-string = "a disjunctive string"

Flag = {T, NIL}

Formula = "a first order predicate calculus expression"

G-Method = {CONTRA*, DC*, DH*, ES*, FTLA*, NOQUIT*, PEEK*}

G'-Method = {CASES, CONTRA, DC, DH, ES, FTLA, IMPLY*,
             PEEK, QUIT, QUIT*, TRAP*, TRUE}

Goal = Conclusion
       or ('DH Hypothesis Goal)
       or ('DH* Hypothesis <definition of hypothesis> Goal)
       or ('ES (:= a b) Goal) or ('ES* (:= a b) Goal)
       or ('CONTRA* Hypothesis Conclusion)
       or ('CASES Disjunctive-string
                  Disjunctive-string
                  Conclusion)
       or ('BC Conclusion Conclusion)

Heuristics = {FTLA, PEEK,...}

History = "a list of task numbers"

Hypothesis = "a set of hypotheses represented by either a list
             or a conjunctive string"

Message = "a quoted atom or string"

Method-name = G-Method or G'-Method

Method-names = "a list of method names"
```

```
Node = (Task-number Goal History Sigma)

Parity = {T, NIL}

Priority = "an integer or POSINF"

Promoters = {BC, CASES, CONTRA*, DH*, ES*, ->}

Quit = ('Quit Goal Sigma)

Sigma = "a substitution"

SQ = State or Quit

State = (Method-name Goal Sigma)

State' = (Method-name Goal)

States = list of State'

Task = Cons((Method-name Priority), Node)

Task-number = [0..POSINF)

Theorem = "a theorem"

Theorem-label = "a list of symbols"

TNIL = {T, NIL}
```

3.0 FUNCTION DEFINITIONS

AH(G:Goal):Formula

```
If Typep(G,'Contra*) then Cadr(G)
Else if Typep(G,'DH*) then And-on(Caddr(G), AH(Caddr(G)))
Else if Typep(G,'ES*) then AH(Caddr(G))
Else if Typep(G,'->) then And-on(Cadr(G), AH(Caddr(G)))
Else if Typep(G,'&) then AH(Cadr(G))
Else NIL
```

APPLY-HEURISTIC(H:Hypothesis, M:Method-name, N:Node, P:Parity):NIL

```
If M = 'FTLA then FTLA(N,P)
Else if M = 'PEEK then PEEK(H,Conclusion1(N),N)
    .
    . calls to other heuristics
    .
Else NIL
```

APPLY-METHOD(M:Method-name, N:Node, P:Parity):NIL

```
1. Q <- Apply-Method*(M,N,P)

2. If Q = NIL then NIL
   (** A method which does not call IMPLY returns NIL **)

3. Else if M = 'IMPLY* and Conclusion1(Q) = Conclusion1(N) then NIL

4. Else if Quitter(Q) then

    <if ~Typep(Conclusion1(Q),'BC) then Generate-tasks(Q,N)
     else
          (** see section on QUIT-LT **)
          Noquit-tasks(Cadr(Conclusion1(Q)),
                       Caddr(Conclusion1(Q)),
                       Caddr(Q),
                       N),

     if ~(Sigma(Q) = T) then (** see section on trapping **)
        for each u in Sigma(Q) do
            New-task(('Trap* Goal(N) (u)), N)>

5. Else New-task((TRUE Goal(N) Car(Q)), N)
   (** add the "TRUE-task" to the AGENDA **)
```

APPLY-METHOD*(M:Method-name, N:Node, P:Parity):Ans

```
If M = 'Contra then Pbc(Conclusion1(N), N)
Else if M = 'DC then Define-conclusion(Conclusion1(N), N)
Else if M = 'DH then Define-hypothesis(Cadr(Goal(N)), N)
Else if M = 'ES then Equality-substitution(N)
Else if Memq(M, Heuristics)
     then Apply-Heuristic(New-H(H,Goal(N)), M, N, P)
Else (** M must be a method name ending in '*' **)
     Callimply(M, Goal(N), N)
```

C-METHODS(Q:Quit):Method-names

This function looks at CONCLUSION1 of Q and returns a list of method names which may be useful in proving it. If no available method applies to the root node, '(IMPLY*) is returned, indicating that IMPLY should be tried.

C-METHODS-FIRST(H:Hypothesis, C:Conclusion):TNIL

This function returns T if methods which apply to C should be tried before any methods which apply to H. Its domain dependence has a great influence on the search conducted by the system (see the section on priorities in chapter 4 of [1]). It is called by GENERATE-TASKS before attempting to generate tasks based upon methods which apply to the hypothesis.

CALLIMPLY(M:Method-name, G:Goal, N:Node):Ans

```
0. If TRACEOVERD then (** running completely automatically **)
     Print some message indicating which task has
     been selected from the AGENDA

1. EXCLUDE <- NIL
   QUIT-LT <- T

2. If M = 'Trap* then EXCLUDE <- Cadddr(N)
   (** EXCLUDE set to sigma part of node N **)

3. IMPLY(DB, H, G, TL, 'B, NIL)
```

CHOOSESUBST(H:Hypothesis, C:Conclusion,
A,B:Formula):Direction

This function returns

(i) (A B) if it decides B should be substituted for A

(ii) (B A) if it decides A should be substituted for B

(iii) else NIL

CONCLUSION1(S:SQ):Formula

Conclusion1*(Goal(S))

CONCLUSION1*(G:Goal):Formula

This function returns the first subgoal which would have to be proved if IMPLY were called with C = G.

EXAMPLES

G	CONCLUSION1*(G)
A & B	Conclusion1*(A)
A => B	Conclusion1*(B)
('DH Basis(B,A) C)	Conclusion1*(C)
('DH* Basis(B,A) lin ind B & <B> = A C)	Conclusion1*(C)
('ES a:=b C)	Conclusion1*(C)
('ES* a:=b C)	Conclusion1*(C)

```
CYCLE(TH:Theorem, TL:Theorem-label):NIL
---------------------------------------

1. TH <- Skolemize TH

2. H <- Hypothesis of TH
   C <- Conclusion of TH

3. Cycle*(DB, H, C, TL)


CYCLE*(DB:Data-base, H:Hypothesis, C:Conclusion,
       TL:Theorem-label):TNIL
------------------------------------------------

1. GNC <- 0 (** global node count initialization **)
   AGENDA <- NIL
   CLOSED <- NIL

2. Generate-Tasks(('Quit C T), (0 NIL NIL T))
   (** place initial tasks on the AGENDA **)

3. Overdirector


DEFINE-CONCLUSION(C:Conclusion, N:Node):NIL
-------------------------------------------

TEMP <- Definep(C,T)
If ~(TEMP = NIL) then New-task(('DC* TEMP T), N)


DEFINE-HYPOTHESIS(H:Hypothesis, N:Node):NIL
-------------------------------------------

TEMP <- Definep(H,NIL)
If ~(TEMP = NIL) then
   New-task(('DH* ('DH* H TEMP Conclusion1(N)) T),
            (Car(N) Caddr(Goal(N)) History(N) Cadddr(N)))


DEFINED-TERMS(G:Goal):Formula
-----------------------------

If Typep(G,'DH*) then And-on(Cadr(G), Defined-terms(Cadddr(G)))
Else if Typep(G,'ES*) then Defined-terms(Caddr(G))
Else NIL


EQUALITY-SUBSTITUTION(N:Node):NIL
---------------------------------

New-task(('ES* ('ES* Cadr(Goal(N)) Conclusion1(N)) T),
         (Car(N) Caddr(Goal(N)) History(N) Cadddr(N)))
```

EVLOOP(M:Message):NIL

This function is called from the function Overdirector when TRACEOVERD = NIL, i.e. when running interactively. The message OVERDIRECTOR is printed, and the user is prompted for input. At this point the user may exercise any of the following options.

NAME	SYNTAX	EFFECT
Proceed	line feed	Causes the system to proceed to select and execute the next task on the AGENDA.
Escape	^^	Causes an error exit to the top level of LISP.
Print the AGENDA	A	Causes the AGENDA to be printed.
Print the top task	T	Causes the top task on the AGENDA to be printed.
Print the theorem	TP	Causes the theorem to be printed.
Print the hypothesis	H	Causes the hypothesis to be printed.
Print the conclusion	C	Causes the conclusion to be printed.
Pursue	P n	Causes task n to be pursued by making it the top task on the AGENDA. Will reopen a closed node if necessary.
Successors	S n	Causes the successors of task n which are still on the AGENDA to be printed.
Eval	<expr>	Causes <expr> to be evaluated by LISP.

Most of these options are also available at the interactive stop inside of IMPLY. See [2] for details.

```
GENERATE-TASKS(Q:Quit, N:NODE):NIL
----------------------------------

For each u in C-Methods(Q) do New-task(Cons(u,Cdr(Q)), N)

If ~C-Methods-First(New-H(H,Goal(Q)), Conclusion1(Q)) then
   for each u in H-Methods(Q) do New-task(Append(u,'(T)), N)


GOAL(S:SQ):Goal
---------------

Cadr(S) (** this function points to the SUBR for CADR **)


HISTORY(N:Node):History
-----------------------

Caddr(N) (** this function points to the SUBR for CADDR **)


H-METHODS(H:Hypothesis, Q:Quit):States
--------------------------------------

H-Methods*(New-H(H, Goal(Q)), Conclusion1(Q), Goal(Q))
```

H-METHODS*(H:Hypothesis, C:Conclusion, G:Goal):States

This function looks at each hypothesis in H individually and decides whether instantiating its definition may be useful. If a hypothesis is an equality, CHOOSESUBST is also called to determine whether it may be useful for equality substitution. A list is returned, each element of which has the form (M (M E G)) where M is either 'DH or 'ES and E is

(i) a hypothesis whose definition may be instantiated if M = 'DH

(ii) (:= a b) if M = 'ES and CHOOSESUBST has decided to try to substitute b for a by looking at the equality a = b in H.

NEW-GOAL(M:Method-name, G',G:Goal):Goal

```
If Typep(M,G'-Method) then G'
Else Typep(M,G-Method) then Subst(G', Conclusion1*(G), G)
Else if Memq(M,BC-lemmas)
     then Subst(('=> Eval(M) Conclusion1*(G')), Conclusion1*(G'), G')
  (** this is for the backward chaining heuristic discussed
      in chapter 4 of [1] **)
Else ERROR
```

NEW-H(H:Hypothesis, G:Goal):Hypothesis

1. D <- Andatom(Defined-terms(Conclusion1*(G)))

2. Return Set-Diff(Andatom(And-on(H, AH(G))), D)

NEW-TASK(S:State, N:Node):NIL

1. M <- Car(S) (** method name **)

2. P <- Priority(M, S, History(N))

3. GNC <- GNC + 1

4. G <- New-goal(M, Goal(S), Goal(N))

5. H' <- Cons(Car(N), History(N))
   (** new task is a descendent of N **)

6. Sort-in(((M P) GNC G H' Sigma(S)))
   (** sort the new task into the AGENDA **)

NOQUIT-TASKS(C:Conclusion, G:Goal, S:Sigma, N:Node):NIL

```
New-task(('Noquit* ('BC C if Typep(G,'BC) then Cadr(G)
                          else Conclusion1(G)), N)

If ~Typep(G,'BC)
    then Generate-Tasks(('Quit Subst(G,Conclusion1(N),Goal(N)) S), N)

If Typep(G,'BC) then Noquit-tasks(Cadr(G), Caddr(G), S, N)
```

The recursion is necessary, because back chaining may have been tried more than once before it failed. For example, when attempting to prove

R & (D & E => P) & (P => C) & (Q => C) & (R => C) => C & R

IMPLY will return

('Quit (& ('BC C ('BC P D & E)) R) T)

and two "NOQUIT-tasks" whose goals are ('BC C P) and ('BC P D) will be added to the AGENDA.

OVERDIRECTOR:TNIL

```
0. If ~TRACEOVERD then Evloop('OVERDIRECTOR)
   (** TRACEOVERD = T means running completely automatically **)

1. If the AGENDA is empty then print ** No proof found **
   and return NIL

2. J <- Select-task(T)

3. Apply-method(Caar(J), Cdr(J), T)

4. If the AGENDA is not empty and the first task on the AGENDA is
       the "TRUE-task" then print ** Proof Found ** and return T
   Else Goto 0
```

PBC(C:Conclusion, N:Node):NIL

```
If Typep(Conclusion*(Goal(N)),'Contra*) then NIL
   (** only allow proof by contradiction once **)
Else
   X <- Pbc*(And-on(H, AH(Goal(N))), NIL)
   Y <- Reduce(('\ C))
   For each q in X do
       New-task(('CONTRA* ('CONTRA* Y Reduce(('\ q))) T), N)
```

PBC*(H:Hypothesis, L:Hypothesis):Hypothesis

This function returns in a list each hypothesis which might be contradicted. It is very domain dependent. For example, in the domain of linear algebra it is not useful to try to contradict a hypothesis such as A el R(M,N), whereas it is sometimes useful to try to contradict a hypothesis such as lin ind A.

PRIORITY(M:Method-name, S:State, H:History):Priority

```
NW <- -10 * Length(H) (** negative weighting term **)

NW +
   if M = 'DH then Priority*(M, Cadr(Goal(S)))
   else if M = 'ES then 0
   else if M = 'FTLA then 100
   else if M = 'IMPLY* then 100
   else if M = 'Noquit* then 0
   else if M = 'Trap* then 100
   else if M = TRUE then 10000
      .
      .  other clauses
      .
   else Priority*(M, Conclusion1(S))
```

The negative weighting term tends to maintain a breadth first search by lowering the priority proportional to the number of predecessors of the node being expanded. In fact, if all priorities are identical, then negative weighting will maintain a breadth first search. The assignment of different priorities is thus a means of dictating the search strategy.

PRIORITY*(M:Method-name, E:Formula):Priority

This function looks at the method name M and the formula E to which M is to be applied and returns an integer, which is the priority of applying M to E. Priorities returned are between -100 and 100; however, tasks may receive priorities less than the static priority returned here, because of the negative weighting term (see Priority).

QUITTER(X:Ans):TNIL

```
Eq(Car(X),'Quit)
```

SELECT-TASK(F:Flag):Task

```
If TRACEOVERD then (** running completely automatically **)
   <J <- Car(AGENDA),
    AGENDA <- Cdr(AGENDA)
    return J>

Else
   <J <- first task on AGENDA whose node does not appear on CLOSED,
    If F = T then CLOSED <- Cons(Cdr(J), CLOSED),
    return J>
```

The CLOSED list is maintained for interactive convenience. When running interactively and debugging it is sometimes necessary to reopen a node which has previously been closed.

SIGMA(S:SQ):Sigma

```
Caddr(S)  (** this function points to the SUBR for CADDR **)
```

SORT-IN(T:Task):NIL

This function inserts the task T into the AGENDA based on its priority. It calls SORT-IN* to do the insertion.

4.0 THE INTERFACE BETWEEN THE OVERDIRECTOR AND IMPLY

From the viewpoint of the OVERDIRECTOR, the interface with IMPLY is contained entirely within the function CALLIMPLY. From the viewpoint of IMPLY, the interface cannot be localized to any one function. Recall that IMPLY is charged with proving the total subgoal sent to it, and, failing this, it must return to the OVERDIRECTOR that portion of the subgoal which it was not able to prove. This portion will often be the entire subgoal; however, there are cases in which partial proofs may be obtained. In either case, the portion returned by IMPLY is collected by the functions AND-C, OR-H, PROMOTE, and TRYBACKCHAINING.

With the exception of the function PROMOTE, these are the functions which introduce new nodes in the proof tree which IMPLY is exploring. It is therefore not suprising that they should be the ones responsible for reporting the success or failure of the exploration. The function PROMOTE is included because of its role as an interpreter of the goal sent to IMPLY. This will be made clear below. All four of these functions call on the function OVERD to combine remaining subgoals which they can "see" with subgoals whose failure has been reported to them.

```
AND-C(EXCLUDE:Sigma):Ans
========================

(** Goal: H -> A & B **)

If [H -> A]Fix-pv(A,B) returns X then

   if Quitter(X) then return Overd(X,BX,T)

   else if [H -> BX]Fix-pv(A,B) returns Y then

      if Quitter(Y) then return Overd(Y,NIL,X)

      else if Compose(X,Y) = NIL then return NIL

      else if Compose(X,Y) intersect EXCLUDE = NIL
           then return Compose(X,Y)

      else
           <TEMP <- Select-intersect(EXCLUDE,X,Y)
            return AND-C(EXCLUDE union {TEMP})>

    else return NIL

else return NIL
```

EQSUB*(A,B:Formula):NIL

```
(** called only by PROMOTE **)
1. DB <- Subst(A,B,DB)
2. AH <- Subst(A,B,AH)
3. H <- Subst(A,B,H)
4. G <- Subst(A,B,G)
5. Return NIL
```

OR-H(EXCLUDE:Sigma):Ans

```
(** Goal: A v B => C **)

If [A => C]Fix-pv(A,B) returns X then

   if Quitter(X) then return
      Overd('(Quit NIL T), ('Cases A v B, A v B, C), Caddr(X))

   else if [BX => C] returns Y then

      if Quitter(Y) then return
         Overd('(Quit NIL T), ('Cases B, BX, C), X)

      else if Compose(X,Y) = NIL then return NIL

      else if Compose(X,Y) intersect EXCLUDE = NIL then
           return Compose(X,Y)

      else
           <TEMP <- Select-intersect(EXCLUDE,X,Y)
            return OR-H(EXCLUDE union {TEMP})>

   else return NIL

else return NIL
```

OVERD(Q:QUit, C:Conclusion, S:Sigma):Quit

```
1. G <- And-on(Goal(Q), C)
2. S' <- Sigma(Q) union S
3. Return ('Quit G S')
```

```
PROMOTE(DB:Data-base, AH,H:Hypothesis, G:Goal,
      TL:Theorem-label):Ans
------------------------------------------------

If Typep(G,'BC) then (** see section on QUIT-LT **)
   <QUITGOAL <- Conclusion1*(Caddr(G)),
    QUIT-LT <- NIL,
    X <- Promote*(DB, AH, H, Cadr(G)),
    if Quitter(X) then
       Overd('(Quit NIL T), Subst(Goal(X),G,C), Sigma(X))
    else X>

Else if Typep(G,'Cases) then (** see OR-H **)
   Promote(DB, Remaining-cases(Cadr(G), Caddr(G), AH),
               Remaining-cases(Cadr(G), Caddr(G), H),
               Caddr(G),
               TL)

Else if Typep(G,'DH*) then
(** EXAMPLE G = (DH* Basis(B,A) (1in ind B & <B> = A) G') **)
   Promote(DB, And-on(Caddr(G),AH), H, Cadddr(G), TL)

Else if Typep(G,'ES*) then
   (** EXAMPLE G = (ES* (:= a b) G') **)
   <H' <- Copy(H),
    Eqsub*(Caddadr(G), Cadadr(G)),
    X <- Promote(DB, AH, H, Caddr(G), Print(Append(TL,'(=S)))),
    if Quitter(X) then
       if Conclusion1(X) = Conclusion1*(C) then
           <H' <- Affected-Hyps(H',H,NIL),
            (** which hypotheses were affected by substitution? **)
            ('Quit if H' = NIL then Caddr(G)
                   else ('-> H' Caddr(G))
                   Sigma(X))>
       else X>

Else if Typep(G,'(Contra*,->)) then
   Promote(DB, And-on(Cadr(G),AH), H, Caddr(G),
           Print(Append(TL,'(P->))))

Else
   <X <- Promote*(DB, AH, H, G),
    if Quitter(X) then
       Overd('(Quit NIL T), Subst(Goal(X),G,C), Sigma(X))
    else X>
```

This function implements what might be called the "generalized rule of promotion." It "interprets" goals as defined in the type declaration section. As a special case, the rule applies to a goal of the type P => Q by adding P to the additional hypotheses, AH, of the subgoal. This is, of course, the usual rule of promotion. After all interpretation has been completed, PROMOTE* is called (where AH will be added to H). If the answer returned by PROMOTE* is one of failure, then OVERD will be called to pass back a failure message together with the (partial) subgoal which remains to be proved.

```
PROMOTE*(DB:Data-base, AH,H:Hypothesis, C:Conclusion):Ans
----------------------------------------------------------

Imply(DB, And-on(AH,H), C, TL, if QED-LT then 3 else 'B, PV)
PROMOTION():Ans
---------------

Promote(DB, NIL, H, C, TL)
```

This function is called by IMPLY whenever the main predicate of the conclusion is a member of the type Promoters.

```
REMAINING-CASES(D,DS:Disjunctive-string,
                H:Hypothesis):Hypothesis
-----------------------------------------

(** called only by PROMOTE **)
If Typep(H,'&) then
   And-on(Remaining-cases(D,DS,Cadr(H)),
          Remaining-cases(D,DS,Caddr(H))

Else if Typep(H,'v) then
     <B <- Oratom(H)
      U <- B  Oratom(D)
      if B = U then return H
      else return DS>

Else H
```

```
TRYBACKCHAINING(EXCLUDE:Sigma):Ans
----------------------------------

(** Goal: B => C where B is an implication **)

If TRYB(B,C) returns (X, H', C') then

   if [H => H'X]Fix-pv(H',C') returns Y then

      if Quitter(Y) then
         return Overd(('Quit ('BC C Goal(Y)) Sigma(Y)), NIL, X)

      else if Compose(X,Y) = NIL then return NIL

      else if Compose(X,Y) intersect EXCLUDE = NIL then
           return Compose(X,Y)

      else
           <TEMP <- Select-intersect(EXCLUDE,X,Y)
            return TRYBACKCHAINING(EXCLUDE union {TEMP})>

   else return NIL

else return NIL
```

5.0 THE TRAPPING PROBLEM

Since IMPLY scans the hypothesis of a theorem from "left to right" it is susceptible to falling into traps by making the wrong substitution for a variable. For example, consider the theorem

(*) Pa & Pb & Qb => Some x (Px & Qx).

IMPLY will quit on (*) returning ('Quit Qa {a/x}). By line 3.3 of Apply-method, this will cause a "TRAP-task" to be added to the AGENDA, since the substitution part of the returned answer is not T. If this "TRAP-task" is ever pursued, then Callimply will set EXCLUDE <- {a/x} which will permit the correct substitution {b/x} to be found.

6.0 THE ROLE OF QUIT-LT

When the OVERDIRECTOR calls IMPLY to attempt to prove a subgoal, it permits IMPLY to work until it encounters a subgoal which it cannot prove. At this point IMPLY will transfer control back to the OVERDIRECTOR, passing back the subgoal upon which it quit working. It does so, anticipating that the OVERDIRECTOR will be able to suggest some method which will enable IMPLY to prove the subgoal the next time it is tried. But the failure of IMPLY may lie in the fact that it was tring to prove an unprovable subgoal. For example, consider trying to prove the theorem

(1) R & (P -> C) & (Q -> C) & (R -> C) -> C.

By back chaining, the subgoal

(2) R & (P -> C) & (Q -> C) & (R -> C) -> P

is set up and IMPLY quits, returning ('Quit P T) to the OVERDIRECTOR. Lacking any methods which might be useful in proving P, this theorem is unprovable by the system described so far. Of course, the problem here lies in using the wrong hypothesis, not in the lack of methods for proving P.

What we need is a method which will allow us to continue the search for proof. Simply recalling IMPLY on (1) will serve no purpose because it will quit returning the same answer as before. Calling IMPLY on (2) will also not work, because we already know that IMPLY cannot prove P. The solution is to recall IMPLY on (1), allowing it to continue past the first subgoal which it cannot prove and returning NIL as its answer.

The method which does this is called NOQUIT*. It is invoked when, as in the example, back chaining has forced consideration of a subgoal which IMPLY is unable to prove. In the example, Trybackchaining will return ('BC C P) as part of its answer when IMPLY fails to prove the hypothesis P. Apply-method (line 3.2) will then add the "NOQUIT-task" whose goal is ('BC C P) to the AGENDA, to allow for the possibility that a back chaining trap has occurred. If this task is later pursued, Promote will set QUITGOAL <- P and QUIT-LT <- NIL. When P again becomes the conclusion by back chaining, Provec will return NIL, thus causing Trybackchaining to return NIL and allowing the next hypothesis to be considered. A description of Provec follows.

PROVEC():Ans

```
Above

If K > Upperk then
   <Print C, Print "Provec failed",
    if QUIT-LT = T then return ('Quit C T)
    else return NIL>

Else if QUIT-LT = NIL and C = QUITGOAL then
   <QUIT-LT <- T,
    Print "Quitting on", Print C,
    return NIL>

Else if ~Memq(K,Provec-levels) then <K <- K+1, Goto Above>

Else if K=i and Proveci returns X then return X

Else <K <- K+1, Goto Above>
```

Clause 4 actually represents several clauses, one for each value of i for which a function called Proveci has been defined. This allows for several different "levels of proof strategy." The variable Upperk is a bound on the levels to be tried on this call to IMPLY. K is initialized to Lowerk inside of the function IMPLY before Provec is entered. The variable Provec-levels can be used as a mask to disable any given levels before IMPLY is called.

7.0 REMARKS

The semi-formal specification of the OVERDIRECTOR presented in this document is intended to be of use to those who may wish to implement this system or one similar to it. The document is not self-contained, as it contains references to functions which are not themselves described. This defect is partially remedied by other documents [2,3]. A complete remedy presupposes the existence of a static program, a state which few research programs achieve. The ultimate utility of this document lies in the flexibility of the program it describes. Unfortunately, this flexibility is responsible for modifications which will soon reduce this document to the status of a mere suggestion for the design of future systems.

8.0 REFERENCES

1. Peter Bruell. An AGENDA Driven Theorem Prover. Ph.D. Thesis, The Univ. of Texas at Austin, August 1979.

2. Peter Bruell. The Control Structure of IMPLY. The Univ. of Texas at Austin Math. Dept. Memo ATP-45, August 1978.

3. W. W. Bledsoe and Mabry Tyson. The UT Interactive Theorem Prover. The Univ. of Texas at Austin Math. Dept. Memo ATP-17a, June 1978.