EVALUATION OF LOGIC PROGRAMS BASED ON NATURAL DEDUCTION

Seif Haridi and Dan Sahlin
Department of Telecommunication and Computer Systems
Royal Institute of Technology
Stockholm, Sweden

(D R A F T)

0. Introduction

In this paper we show how a large subset of first order logic can be reasonably efficienty interpreted. Logic programming has usually been restricted to a conditional type of statements called Horn clauses. General logical statements that are natural to write cannot directly be expressed by Horn clauses. This is due to the fact that Horn clauses express only the "if-halves" of "iff-definitions". This has meant that this that are easily expressed in first order logic has been done in meta-logic. For example, negation has been treated as nonprovability and special 'setof' contructs have been devised to find all the solutions to a relation.
To solve these problems we construct an abstract machine called 'gepr' (=goal, environment, program and resumption register). The ideas of the 'gepr' machine resemble the 'secd' machine for functional programming languages [La63][He80], which describes what state transitions that are allowed. Although we have used a version of Horn clauses to describe the state transitions, they could equally well have been described in an imperative language.
The basis for the interpreter are the rules of a natural deduction system as shown in [Ha81].

1. Sample programs

A program consists of a set of relation definitions, where a relation is a predicate-logic statement of one of the forms:

1. relation_name($term_1$,$term_2$,...,$term_n$) <-> arbitrary logic statement

2. relation_name($term_1$,$term_2$,...,$term_n$) -> arbitrary logic statement

3. relation_name($term_1$,$term_2$,...,$term_n$) <- arbitrary logic statement

4. ¬ relation_name($term_1$,$term_2$,...,$term_n$)

We also apply a rule of implicit quantification for the variables not being quantified:

> Variables occuring in the "head" of the relation are universally quantified over the whole statement, while the other variables are existentially quantified over the right hand side of the relation definition.

For example
```
     list(w) <-> w=[] or w=[x|y] & list(y)
```
actually means in logic
```
     ∀w( list(w) <-> ∃x∃y (w=[] or w=[x|y] & list(y)) )
```

To be able to interpret the statement above, we break it down further into the conjunction of two statements:

```
     ∀w( list(w) -> ∃x∃y (w=[] or w=[x|y] & list(y)) )
```
and
```
     ∀w( list(w) <- ∃x∃y (w=[] or w=[x|y] & list(y)) )
```

The fourth type of statement given above, the negation, is also transformed into an implication. For example,
```
     ¬ member(x,[]).
```
is transformed to
```
     member(x,[]) -> False.
```

Some more examples:

The full definition set of 'member':

```
¬ member(x,[]).
member(x,[y|z]) <-> x=y or member(x,z).
```

The predicate 'class' tests if all members of 'sl' take course c, or for a certain course finds all its members etc.

```
class(c,sl) <-> ∀s (takes_course(s,c) <-> member(s,sl)).

takes_course(x,y) <-> x=D & y=C1 or
                      x=J & y=C1 or
                      x=J & y=C3.

maths_course(z) <-> z=C1 or z=C3.
```

A "maths major" is a person who takes all maths courses:

```
maths_major(x) <-> ∀y (maths_course(y) -> takes(x,y)).
```

This enables us to find out that maths_major(J) is true, or even makes it possible to find all "maths majors" with a variant of the 'class' predicate above.

2. Types

To simplify the description of the interpreter below, we here introduce a type concept in first order logic. What we actually do is that we have a convenient way to define relations that are true iff their arguments are in a certain domain. We will not here try to make an exact definition of the transformation between our simplified notation and logic, but just show a few examples.

Having the type definition

```
TYPEDEF formula = And(formula,formula) ! Or(formula,formula) ! Eq(term,term)
```

we get the corresponding logic statement

```
formula(x) <-> ∃y∃z(x=And(y,z) & formula(y) & formula(z) ) or
               ∃y∃z(x=Or(y,z)  & formula(y) & formula(z) ) or
               ∃y∃z(x=Eq(y,z)  & term(y) & term(z)).
```

To be able to define a list of a certain type

```
TYPEDEF list(t) = [] ! [t|list(t)]
```

we find it convenient to use schemas in first order logic:

```
list(t)(x) <-> x=[] or ∃y∃z( x=[y|z] & t(y) & list(t)(z) ).
```

(A schema can take a relation as an argument).
Obvious abbreviations are also used in the type definitions.

3. Formulas

The full type definition of a formula looks like this

```
TYPEDEF formula = And(formula,formula) ! Or(formula,formula) !
                  Imp(formula,formula) ! False !
                  Eq(term,term) ! Rel(name,list(term)) !
                  All(list(name),formula) ! Exist(list(name),formula).

TYPEDEF term = Var(name) ! Dstruct(name,list(term))
```

and the relation 'name' is appropriately defined.

For example, the formula "∀y (maths_course(y) -> takes(x,y))" has a corresponding abstract tree:

```
All(['y'],
    Imp(Rel('maths_course',[Var('y')]),
        Rel('takes',[Var('x'),Var('y')])))
```

The formula "∀s (takes(s,c) <-> member(s,sl))" is first split into the conjunction of two formulas

"∀s ((takes(s,c) -> member(s,sl)) & (member(s,sl) -> takes(s,c)))"

which has the corresponding abstract tree

```
All(['s'],
    And(Imp(Rel('takes',[Var('s'),Var('c')]),
            Rel('member',[Var('s'),Var('sl')])),
        Imp(Rel('member',[Var('s'),Var('sl')]),
            Rel('takes',[Var('s'),Var('c')]))))
```

4. A program

The type definitions are extended by

```
TYPEDEF program = list(relation)

TYPEDEF relation = Reldef(name,backdef,forwarddef).

TYPEDEF backdef = list( assertion ! limplication ).

TYPEDEF forwarddef = list(rimplication).

TYPEDEF assertion = Assert(list(name),list(term)).

TYPEDEF limplication = Limp(list(name),list(term),formula)

TYPEDEF rimplication = Rimp(list(name),list(term),formula)
```

Each relation definition definition set is split into two parts, one for the forward implications (->) and one for the backward implications (<-). This corresponds to the types 'forwarddef' and 'backdef' above, each consisting of a list of assertions or implications.
The relation 'member'

```
        ¬ member(x,[]).
        member(x,[y|z]) <-> x=y or member(x,z).
```

is transformed to

```
[Reldef('member',[Limp(['x','y','z'],
                       [Var('x'),Dstruct('.',[Var('y'),Var('z')])],
                       Or(Eq(Var('x'),Var('y')),
                          Rel('member',[Var('x'),Var('z')]))),
                  [Rimp(['x'],
                       [Var('x'),Dstruct('[]',[])],
                       False),
                   Limp(['x','y','z'],
                       [Var('x'),Dstruct('.',[Var('y'),Var('z')])],
                       Or(Eq(Var('x'),Var('y')),
                          Rel('member',[Var('x'),Var('z')])))])]
```

5. The environment

The values of the variables during a computation are found in the environment.

```
TYPEDEF environment = list(<Context(loc,context),loc>)

TYPEDEF context = list(Binding(name,value))

TYPEDEF value = <term,loc> ! STAR ! UNBOUND
```

where loc is an integer.

A formula is always considered in a certain context, and this context is conveniently referred by a location (an integer).

If we have the formula

$\exists x,y,z\ (\ \exists z,x\ (\ q(x,y,z) \rightarrow r(x,y)\ )\ \&\ s(x,z)\ \&\ p(x,y,z))$

we will in a procedural interpretation get the following:
After having "performed" the outermost existential quantifier the variables x,y and z are known. We then have

[]	
x	UNBOUND
y	UNBOUND
z	UNBOUND

10 [<Context([], [Binding('x',UNBOUND), Binding('y',UNBOUND), Binding('z',UNBOUND)]),10>]

and the current context is 10.
If we then immediatly perform the inner quantifier we get

10	
z	UNBOUND
x	UNBOUND

11 [<Context(10, [Binding('x',UNBOUND), Binding('y',UNBOUND)]),11>,

[]	
x	UNBOUND
y	UNBOUND
z	UNBOUND

10 <Context([], [Binding('x',UNBOUND), Binding('y',UNBOUND), Binding('z',UNBOUND)]),10>]

The value of variable 'x' in context 11 we quite naturally find in context 11, but the variable 'y' can't be found in context 11. We then follow the **static chain** which is given by the first argument of a context. The static chain of context 11 is 10, while context 10 does not have any static chain. We have a special notation for the value of a variable in a certain

context; the variable 'x' in context l1 is written <Var('x'),l1>.

6. The 'gepr' machine

The basis for the 'gepr' machine is a state transition system. We have a set of state transition rules

$$state_i \dashrightarrow state_{i+1}$$

The behaviour of the machine is described by the transitive closure of the transition relation, which we write as '-->*' and is defined by

$$terminal\text{-}state \dashrightarrow^{*} terminal\text{-}state$$

$$state \dashrightarrow^{*} terminal\text{-}state \text{ if } state \dashrightarrow^{*} state_1 \text{ and } state_1 \dashrightarrow terminal\text{-}state$$

Each 'state' consists of four 'registers':

G Goal-stack
E Environment
P Program
R Resumption register to handle backtracking

We have already shown the types of E and P, and R will be elaborated when we come to 'or' in formulas.
The goal-stack is perhaps the most complex type of the 'gepr'-machine. It contains information in very goal directed manner of what that has to be done. Since the machine has two major modes of execution, backward proof and forward proof mode, the goal-stack contains two types of items. (The special "goal" Fail may also occur on the goal-stack).

```
TYPEDEF goalstack = list(goal)
TYPEDEF goal = B(<formula,loc>, conclusion_environment) !
               F(list(<formula,loc>), <formula,loc>, conclusion_environment) !
               Fail
TYPEDEF conclusion_environment = ...almost the same as environment...
```

The backward proof mode of execution corresponds roughly to the normal "Prolog" mode of execution, while the forward mode is needed to handle implications in formulas.
We start with explaining the backward proof mode, but first we show how to initialize the 'gepr' machine.

If we want to evaluate a formula 'f' in a program 'p', the 'gepr'-machine starts with

```
        G                E   P  R
([B(<f,[]>,[])], [], p, [])
```

i.e. the goal-stack G contains just the item B(<f,[]>,[]). This means that we are going to perform a **backward proof** of f in an empty context. The context is empty because we assume that all variables are explicitly quantified in the formula, and contexts will be created for those variables.
The second argument of B (the conclusion_environment) is also empty initially.
The environment is empty since we don't have any bindings of any variables when we start.
The resumption register is empty since we have no backtracking points.

A successful final state of the 'gepr' machine is

```
([], e, p, r)
```

where the bindings of the variables are found in the environment e. If we, for some reason, want another solution, the machine may be restarted again with the information in the resumption register. This is shown below in the 'Fail'-transition.
It may also happen that the whole computation fails. This is the case if the final state is

```
([Fail|gs], e, p, [])
```

We are now ready to show the state transition rules, i.e. rules that convert one state of the 'gepr' machine to the next. Each rule corresponds to a rule in a natural deduction system.

7. Backward proof

7.1. And

If we in a natural deduction system are going to prove "f1 & f2", we first prove f1 separately and then prove f2. Due to the '&' introduction rule we have

```
   f1   f2
  ---------
   f1 & f2
```

which is read backwards in a backward proof. In the 'gepr' machine this corresponds to the state transition

```
([B(<And(f1,f2),l>,ce)|gs],      e, p, r)  -->
([B(<f1,l>,ce),B(<f2,l>,ce)|gs], e, p, r)
```

Since no new variables are introduced, the context 'l' is unchanged. The rest of the goal-stack is 'gs', and is left untouched by the transition.

7.2. Or

If we want to prove "f1 or f2" we may either prove f1 or f2:

```
      f1                      f1
  ----------     or       ----------
   f1 or f2                f1 or f2
```

And the corresponding transition

```
([B(<Or(f1,f2),l>,ce)|gs],e,p,r) -->
([B(<f1,l>,ce)|gs]       ,e,p,[GEP([B(<f2,l>,ce)|gs],e,p)|r])
```

The computation continues with a backward proof of f1, while we have saved the contents of the registers G, E and P on the resumption stack for the alternative computation. If we, for any reason, fail with proving f1, we may retry and prove f2. A failure of a computation is indicated by a special "goal" called 'Fail' on the goal-stack. Although we have not yet shown how a 'Fail' gets to the goal-stack, we here show how the 'gepr' machine reacts.

```
([Fail|gs],e, p,[GEP(gs1,e1,p1)|r1]) -->
(gs1,      e1,p1,r1).
```

It may however occur that the resumption stack is empty. We then don't have any valid alternatives and the whole computation has failed.

```
([Fail|gs],e,p,[])      no solution!
```

The type of the resumption stack is called 'dump':

```
 TYPEDEF dump = list(GEP(list(goal),environment,program))
```

7.3. False

The simplest way to fail in a backward proof is to find an explicit 'False' in the goal-stack. The 'gepr' machine simply converts this to Fail.

```
([B(<False,_>,_)|gs],e,p,r) -->
([Fail],             e,p,r).
```

7.4. Exists

If a group of variables are existentially quantified, we allocate a storage for those variables and initialize them to unbound. This will effect the environment. In natural deduction this becomes

```
   f(v)
--------
∃v(f(v))
```

and the proof may continue backwards from 'f(v)'.

```
([B(<Exist(vs,f),l>,ce)|gs],e,p,r) -->
([B(<f,l1>,ce)|gs],         e1,p,r)   if  newenv(vs,l,e,UNBOUND)=[l1,e1].
```

where e1 is the new environment and l1 is the location of the new context. The type of the function newenv is

```
    TYPEOF newenv(list(name),loc,environment,value)=[loc,environment]
```

For example, the first environment shown on page 5 could have been created by newenv(['x','y','z'],[],[],UNBOUND).

7.5. For all

A group of variables may alternatively be universally quantified. In the natural deduction system we mark the variables (with a star) so they cannot become bound.

```
   f(*v)
----------
∀v(f(v))
```

The 'gepr' machine makes almost the same things as for an existential quantification, but all the variables are bound to the special value 'STAR'.

```
([B(<All(vs,f),l>,ce)|gs],e,p,r) -->
([B(<f,l1>,ce)|gs],       e1,p,r)   if  newenv(vs,l,e,STAR)=[l1,e1].
```

When such a variable is found during a unification, it cannot be bound, and

will remain having the value 'STAR'. This is quite natural since a universally quantified variable cannot be restricted to a special value.

7.6. Equality

The special atomic relation '=' also gets a special treatment. In a backward proof however, the treatment seems to be quite normal. We just invoke unification:

```
We have two cases
    ([B(<Eq(t1,t2),l>,ce)|gs],e,p,r) -->
    gepr(gs,e1,p,r)
        if  unify([<t1,l>],[<t2,l>],ce,e) = e1 and
            e1 /= Fail
or
    ([B(<Eq(t1,t2),l>,ce)|gs],e,p,r) -->
    gepr([Fail],e,p,r)
        if  unify([<t1,l>],[<t2,l>],ce,e) = e1 and
            e1 = Fail
```

The function 'unify' returns the new environment in case of success, otherwise it returns the constant 'Fail'. The type of 'unify' is

```
TYPEOF unify(list(<term,loc>),list(<term,loc>),
            conclusion_environment, environment) = (environment ! Fail)
```

The unification differs from a standard unification in several ways.
When we want to get the value of a variable, we always first look in the ce ('conclusion_envirionment') for reasons which will be explained later. If the variable is unbound there we then use the normal environment.
As already mentioned, the STAR variables are not allowed to be bound to be bound to anything, and no variable is allowed to become bound to a STAR variable. There is however one exception when we can convert a proof made with a binding to a STAR variable to a proof without such a binding. In natural deduction we may have

```
      Π
      :
    B(x,*y)
  -----------
  ∃x B(x,*y)
```

where B(x,*y) is an arbitrary complex formula containing x and y. If we have to assume x=*y in order to perform the proof Π, we can convert the whole proof Π to a new proof that does not need that assumption. Since

```
      Π2
      :
    B(*y,*y)
  -----------
  ∃x B(x,*y)
```

is a valid step in natural deduction, and the proof Π2 does not contain any assumptions on *y, the formula "∃x B(x,*y)" is valid. What is crucial is that the existentially quantified variable must 'declared' after the universally quantified variable (the 'STAR' variable).
This mechanism allows us to conclude that

```
    ∀x∃y (x=y) is true
```

while

```
    ∃x∀y (x=y) is false
```

Which one of the variables that is 'declared' first is easily tested by comparing the location numbers of the variables. In this case it is important to follow the static chain to get the true location of the variable.

Although not strictly necessary we have also chosen to implement unification so that it can handle cyclic structures [Ha81].

7.7. Atomic relation

In a backward proof when an atomic relation is encounted, we need the program to find the definition of that relation. There may be none, one or several such relation definitions. We take them in the defined order of the backward definition set and perform a unification, which may change the environment. In natural deduction we write

$$\frac{\forall x1,..xn(\ relation\_name(q1,..,qn) \leftarrow f\ ) \qquad \begin{matrix}\Pi \\ : \\ f\end{matrix} \qquad q1=r1\ ..\ qn=rn}{relation\_name(r1,..,rn)}$$

Above the line we have three parts: a part of the relation definition set, the formula f (which is the right hand side of the relation definition), and a group of equalities generated during unification. If are able to prove f in backward mode we may then conclude the formula that is written under the line.
In the 'gepr' machine we first find all the definitions of the relation. This is done by the function 'getbackdef'. For each of the relation definitions found we stack a unification request on the resumption stack. This is done by the function 'newdumpb'. Finally we invoke failure so that the top item (if any) of the resumption stack will be used.

```
([B(<Rel(rn,ts),l>,ce)|gs],e,p,r) -->
([Fail],e,p,r1)              if getbackdef(p,rn) = stmts and
                                newdumpb(ts,stmts,l,ce,gs,e,p,r)=r1
```

Although it looks a bit complicated, the function 'newdumpb' is very simple:

```
TYPEOF newdumpb(list(term),backdef,loc,conclusion_environment,list(goal),
                environment,program,dump)=dump
newdumpb(_,[],_,_,_,_,_,r)=r.
newdumpb(ts,[stm|stms],l,ce,gs,e,p,r) =
    [GEP([B(<Unify(ts,stm),l>,ce)|gs],e,p)|newdumpb(ts,stms,l,ce,gs,e,p,r)].
```

The type of the variable 'stmts' above is 'backdef' (see page 4), which means that we actually have two types of clauses: assertions and relations. We don't show the assertions here since they are identical to a relation with an always true right hand side.
When the 'gepr' machine finds a unification request on top of the goal-stack it first allocates space for the new variables and then performs the unification.

```
We then have two cases:
    ([B(<Unify(ts1,Limp(vs,ts2,f)),l1>,ce)|gs],e,p,r) -->
    ([B(<f,l2>,ce)|gs],                              e2,p,r) )
          if newenv(vs,[],e,UNBOUND) = [l2,e1] and
             unify(ctermlist(ts1,l1),ctermlist(ts2,l2),ce,e1) = e2 and
             e2/=Fail
or
    ([B(<Unify(ts1,Limp(vs,ts2,f)),l1>,ce)|gs],e,p,r) -->
    ([Fail],                                         e2,p,r) )
          if newenv(vs,[],e,UNBOUND) = [l2,e1] and
             unify(ctermlist(ts1,l1),ctermlist(ts2,l2),ce,e1) = e2 and
             e2=Fail

The type of 'ctermlist' is
 TYPEOF ctermlist(list(term),loc)=list(value)

ctermlist([],_) = [].
ctermlist([a|as],l) = [<a,l>|ctermlist(as,l)].
```

7.8. Implies

Finally, here is the rule that changes the execution mode from backward proof to forward proof. In natural deduction we have

```
          f1[1]
          :
          Π
          :
          f2
       --------[1]
        f1->f2
```

The expression under the line is true if we by starting by assuming f1 can prove f2. This proof is marked with Π in the figure above. During that compution certain conclusions may have been drawn, which obviously depend on the assumption f1, and they must therefore be discharged after the subproof. This is schematically indicated by "[1]" in the figure. We solve this problem by having a local 'conclusion environment' for all subcomputations. By this, local conclusions don't effect the global status of the computation (the environment).
In the 'gepr' machine we have

```
    ([B(<Imp(f1,f2),l>,ce)|gs],e,p,r) -->
    ([F([<f1,l>],<f2,l>,ce)|gs],e,p,r)
```

The formula f1 is called the "premise goal". In general we may have a list of "premise goals", but at start there is just one.
We are now ready for

8. Forward proof

The action in forward proof mode generally depends on the form of the premise goal.

8.1. And

The first rule is a simple rewrite

```
([F([<And(f1,f2),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
([F([<f1,l0>,<f2,l0>|fr],<cf,l>,ce)|gs],e,p,r)
```

which extends the list of premise goals.

8.2. Or

If the first premise goal is an 'or'-form we have

```
                                                  f1[1]    f2[1]
                                                  :        :
f1 or f2                                          Π1       Π2
   :                                              :        :
   Π                                 f1 or f2     cf       cf
   :        which we convert to     ------------------------------[1][2]
  cf                                              cf
```

That is, to prove that f1 or f2 implies cf we have to prove that f1 implies cf **and** f2 implies cf. The conclusions drawn at these subcomputations must as usual be removed after the computation.
This means that

```
([F([<Or(f1,f2),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
([F([<f1,l0>|fr],<cf,l>,ce),F([<f2,l0>|fr],<cf,l>,ce)|gs],e,p,r)
```

8.3. Exist

If variables are existentially quantified in the premise goal we get

```
                f(*v)[1]
                  :
                  Π
                  :
   ∃v(f(v))      cf
   ---------------------[1]
          cf
```

We try to perform a proof of 'cf' starting from the premise goal f(*v).
All the existentially quantified variables have the value STAR, and have to follow the rules of a STAR variable. In 'gepr' we get

```
([F([<Exist(vs,f),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
([F([<f,l1>|fr],<cf,l>,ce)|gs],e1,p,r) if newenv(vs,l0,e,STAR) = [l1,e1]
```

8.4. For all

Similarly for universal quantification we have

```
     ∀v(f(v))
    ----------
       f(v)
        :
        Π
        :
       cf
```

and for 'gepr'

```
([F([<All(vs,f),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
([F([<f,l1>|fr],<cf,l>,ce)|gs],e1,p,r) if newenv(vs,l0,e,UNBOUND) = [l1,e1]
```

8.5. False

If the premise is false we can end the subcomputation with success without further computations.

```
 False
-------
  cf
```

And for 'gepr' the computation continues with 'gs':

```
([F([<False,l0>|fr],<_,_>,_)|gs],e,p,r) -->
(gs,                              e,p,r)
```

8.6. Implies

Even in forward proof mode an implication can be found in the premise goal.

```
f1->f2                                  Π1
  :                                     :
  Π       which is converted to         f1        f1-> f2
  :                                     ----------------------------
 cf                                             f2
                                                :
                                                Π2
                                                :
                                                cf
```

We first try to perform the backward proof Π1, and then the forward proof Π2.

```
([F([<Imp(f1,f2),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
([B(<f1,l0>,ce),F([<f2,l0>|fr],<cf,l>,ce)|gs],e,p,r)
```

8.7. Equality

If an equality is found in the premise goal-list, we may use this equality anywhere in the subproof. The situation is very different from ordinary unification, almost the opposite. We first look for the value of a variable in the normal environment (e), and only if the value is UNBOUND or STAR we look in the conclusion environment (ce). If it necessary to bind a variable in order to succeed in the conclusion unification, the new value is only stored in the conclusion environment.
If the conclusion unification fails, it must be remembered that a false premise implies everything, so we have actually succeeded!

```
Two cases:
     ([F([<Eq(t1,t2),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
     ([F(fr,<cf,l>,ce1)|gs],e,p,r)
        if  cf /= <False,_>
        and concunify([<t1,l0>],[<t2,l0>],ce,e) = ce1
        and ce1/=Fail
or
     ([F([<Eq(t1,t2),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
     (gs,                                   e,p,r)
        if concunify([<t1,l0>],[<t2,l0>],ce,e) = ce1
        and ce1=Fail
where
TYPEOF concunify(list(<term,loc>),list(<term,loc>), conclusion_environment,
                 environment) = (conclusion_environment ! Fail)
```

In the code above we check so that the conclusion formula isn't an explicitly 'False' formula. If concunify succeeds we would otherwise be sure that the computation whould fail. To avoid this we test for that special case, and we treat it separately. In this case the only solution for success is to use some sort of 'negative' unification which generates assumptions like 'x≠17'. The advantage of this is a wider domain of executable programs, while the disadvantage is increased nondeterminism. In this paper we will not elaborate this mechanism further.

8.8. Atomic relation

When we find an atomic relation in the premise goal-list, we try to find its definition set in the program. If there are several definitions we take the first and save the rest on the resumption stack. The case in natural deduction when we have found one definition:

$$\frac{\forall x(\ \text{relation\_name}(q1,..,qn) \leftarrow f) \qquad \text{relation\_name}(r1,..,rn) \qquad q1{=}r1,..,qn{=}rn}{\begin{array}{l} f \\ : \\ \Pi \\ : \\ cf \end{array}}$$

Above the line we have three parts: a part of the relation definition set, the premise goal, and a group of equalities generated during unification. We then have to perform the proof Π.
As in backward proof mode we first find the definition set and stack the alternative definitions on the resumption stack. We then invoke failure so the top item on the resumption stack will be used.

```
     ([F([<Rel(rn,ts),l0>|fr],<cf,l>,ce)|gs],e,p,r) -->
     gepr([Fail],e,p,r1)
        if  getforwarddef(p,rn) = stmts and
            newdumpf(ts,stmts,l0,fr,cf,l,ce,gs,e,p,r)=r1
where
 TYPEOF newdumpf(list(term),forwarddef,loc,list(<formula,loc>),<formula,loc>,
                 loc,conclusion_environment,list(goal),environment,
                 program,dump)=dump.
 newdumpf(_,[],_,_,_,_,_,_,_,_,r)=r.
 newdumpf(ts,[stm|stms],l0,fr,cf,l,ce,gs,e,p,r) =
 [GEP([F([<Unify(ts,stm),l0>|fr],<cf,l>,ce)|gs],e,p)|
          newdumpf(ts,stms,l0,fr,cf,l,ce,gs,e,p,r)].
```

In 'gepr' machine we may encounter 'unifications' in forward proof mode. The unification may either fail or succeed.

We have two cases:

```
     ([F([<Unify(ts1,Rimp(vs,ts2,f)),l0>|fr],<cf,l1>,ce)|gs],e,p,r) -->
     ([F(fr,<cf,l1>,ce)|gs],_,_,r)
        if  newenv(vs,[],e) = [l2,e1]
        and unify(ctermlist(ts1,l0),ctermlist(ts2,l2),ce,e1) = e2
        and e2=Fail
or
     ([F([<Unify(ts1,Rimp(vs,ts2,f)),l0>|fr],<cf,l1>,ce)|gs],e,p,r) -->
     ([F([<f,l2>|fr],<cf,l1>,ce)|gs],e2,p,r)
        if  newenv(vs,[],e) = [l2,e1]
        and unify(ctermlist(ts1,l0),ctermlist(ts2,l2),ce,e1) = e2
        and e2/=Fail
```

9. Discussion

It is important to stress that we have not devised a complete theorem prover. This means that there is a domain of formulas that we are unable to prove. We claim however that these formulas usually are not computationally useful. To include them in the set of formulas we can prove would increase the nondeterminism and degrade the system performance.
One improvement towards a more complete system was the 'negative' unification which was discussed on page 14. Since we have not yet been able to use the system for large problems, we not yet sure whether this complication would be worthwile. We do however already know that there is a wide domain of programs where the system has proven to be quite useful.

10. References

[La63] Landin, P J, The Mechanical Evaluation of Expressions, Computer Journal, 6 (4), 308-20, 1963

[He80] Henderson, P, Functional Programming, Prentice-Hall International, 1980

[Ha81] Haridi, S, Logic Programming Based on a Natural Deduction System, thesis, Royal Institute of Technology 1981, Stockholm

[HHT82] Å Hansson, S Haridi, S-Å Tärnlund, Properties of a Logic Programming Language, in Logic Programming edited by K L Clark and S-Å Tärnlund, Academic Press 82