P_{fc} User Manual
version 1.2

Tim Finin
Computer Science and Electrical Engineering
University of Maryland Baltimore County
1000 Hilltop Circle
Baltimore MD 21250
finin@umbc.edu
http://umbc.edu/ finin/

August 1999

Abstract

The P_{fc} system is a package that provides a forward reasoning capability to be used together with conventional Prolog programs. The P_{fc} inference rules are Prolog terms which are asserted as clauses into the regular Prolog database. When new facts or forward reasoning rules are added to the Prolog database (via a special predicate add/1, forward reasoning is triggered and additional facts that can be deduced via the application of the forward chaining rules are also added to the database. A simple justification-based truth-maintenance system is provided as well as simple predicates to explore the resulting proof trees.

1 Introduction

Prolog, like most logi programming languages, offers ba kward haining as the only reasoning s heme. It is well known that sound and omplete reasoning systems an be built using either ex lusive ba kward haining or ex lusive forward haining [5]. Thus, this is not a theoreti al problem. It is also well understood how to "implement" forward reasoning using an ex lusively ba kward haining system and vi e versa. Thus, this need not be a pra ti al problem. In fa t, many of the logi -based languages developed for AI appli ations [3, 1, 6, 2] allow one to build systems with both forward and ba kward haining rules.

There are, however, some interesting and important issues whi h need to be addresses in order to provide the Prolog programmer with a pra ti al, effi ient, and well integrated fa ility for forward haining. This paper des ribes su h a fa ility, P_{fc} , whi h we have implemented in standard Prolog.

The P_{fc} system is a pa kage that provides a forward reasoning apability to be used together with onventional Prolog programs. The P_{fc} inferen e rules are Prolog terms whi h are asserted as fa ts into the regular Prolog database. For example, Figure 1 shows a file of P_{fc} rules and fa ts whi h are appropriate for the ubiquitous kinship domain.

The rest of this manual is stru tured as follows. The next se tion provides an informal introdu tion to the P_{fc} language. Se tion three des ribes the predi ates through whi h the user alls P_{fc}.The final se tion gives several longer examples of the use of P_{fc}

Getting and installing P_{fc}

Look for P_{fc} on ftp. s.umb .edu in /pub/pf /.

2 An Informal Introduction to the P_{fc} language

This se tion des ribes P_{fc} . We will start by introdu ing the language informally through a series of examples drawn from the domain of kinship relations. This will be followed by an example and a des ription of some of the details of its urrent implementation.

Overview

The P_{fc} pa kage allows one to define forward haining rules and to add ordinary Prolog assertions into the database in su h a way as to trigger any of the P_{fc} rules that are satisfied. An example of a simple P_{fc} rule is

```
gender(P,mal
```

```
P, (Q;R), S => T
```

P_{fc} handles su h a rule by putting it into onjun tive normal form. Thus the rule above is the equivalent to the two rules

```
P,Q,S => T
P,R,S => T
```

Bi-conditionals

P_{fc} has a limited ability to express bi- onditional rules, su h as

```
mother(P1,P2) <=> parent(P1,P2), female(P1).
```

In parti ular, adding a rule of the form `P<=>Q` is the equivalent to adding the two rules `P=>Q` and `Q=>P`. The limitations on the use of bi- onditional rules stem from the restri tions that the two derived rules be valid horn lauses. This is dis ussed in a later se tion.

Backward-Chaining P_{fc} Rules

P_{fc} in ludes a spe ial kind of ba kward haining rule whi h is used to generate all possible solutions to a goal that is sought in the pro ess of forward haining. Suppose we wished to define the *ancestor* relationship as a P_{fc} rule. This ould be done as

```
parent(P1,P2) => ancestor(P1,P2).
parent(P1,P2), ancestor(P2,P3) => ancestor(P1,P3).
```

However, adding these rules will generate a large number of assertions, most of whi h will never be needed. An alternative is to define the *ancestor* relationship by way of ba kward haining rules whi h are invoked whenever a parti ular an estor relationship is needed. In P_{fc} this need arises whenever fa ts mat hing the relationship are sought while trying a forward haining rule.

```
ancestor(P1,P2) <= {\+var(P1)}, parent(P1,X), ancestor(X,P2).
ancestor(P1,P2) <= {var(P1),\+var(P2)}, parent(X,P2), ancestor(P2,X).
```

Conditioned Rules

It is sometimes ne essary to add some further ondition on a rule. Consider a definition of sibling whi h states

> Two people are siblings if they have the same mother and the same father. No one an be his own sibling.

This definition ould be realized by the following P_{fc} rule

```
mother(Ma,P1), mother(Ma,P2), {P1\==P2},
  father(Pa,P1), father(Pa,P2)
   => sibling(P1,P2).
```

Here we must add a ondition to the lhs of the rule whi h states the the variables $P1$ and $P2$ must not unify. This is effe ted by en losing an arbitrary Prolog goal in bra es. When the goals to the left of su h a bra keted ondition have been fulfilled, then it will be exe uted. If it an be satisfied, then the rule will remain a tive, otherwise it will be terminated.

Negation

We sometimes want to draw an inferen e from the absen e of some knowledge. For example, we might wish to en ode the default rule that a person is assumed to be male unless we have eviden e to the ontrary

```
person(P), ~female(P) => male(P).
```

A lhs term pre eded by a $\sim$ is satisfied only if *no* fa t in the database unifies with it. Again, the P_{fc} system re ords a justifi ation for the on lusion whi h, in this ase, states that it depends on the absen e of the ontradi tory eviden e. The behavior of this rule is demonstrated in the following dialogue

```
?- add(person(P), ~female(P) => male(P)).
yes
?- add(person(alex)).
yes
?- male(alex).
yes
?- add(female(alex)).
yes
?- male(alex)
no
```

As a slightly more ompli ated example, onsider a rule whi h states that we should assume that the parents of a person are married unless we know otherwise. Knowing otherwise might onsist of either knowing that one of them is married to a yet another person or knowing that they are divor ed. We might try to en ode this as follows

```
parent(P1,X),
parent(P2,X),
{P1\==P2},
~divorced(P1,P2),
~spouse(P1,P3),
{P3\==P2},
~spouse(P2,P4),
{P4\==P1}
  =>
spouse(P1,P2).
```

Unfortunately, this won't work. The problem is that the onjoined ondition

```
~spouse(P1,P3),{P3\==P2}
```

does not mean what we want it to mean - that there is no $P3$ distin t from $P2$ that is the spouse of $P1$. Instead, it means that $P1$ is not married to any $P3$. We need a way to move the qualifi ation `{P3\==P2}` inside the s ope of the negation. To a hieve this, we introdu e the notion of a qualified goal. A lhs term P/C, where P is a positive atomi ondition, is true only if there is a database fa t unifying with P and ondition C is satisfiable. Similarly, a lhs term $\sim P/C$, where P is a positive atomi ondition, is true only if there is no database fa t unifying with P for whi h ondition C is satisfiable. Our rule an now be expressed as follows

```
parent(P1,X),
  parent(P2,X)/(P1\==P2),
  ~divorced(P1,P2),
  ~spouse(P1,P3)/(P3\==P2),
  ~spouse(P2,P4)/(P4\==P1)
  =>
  spouse(P1,P2).
```

Procedural Interpretation

Note that the pro edural interpretation of a P_{fc} rule is that the onditions in the lhs are he ked *from left to ri ht*. One advantage to this is that the programmer an hose an order to the onditions in a rule to minimize the number of partial instantiations. Another advantage is that it allows us to write rules like the following

```
at(Obj,Loc1),at(Obj,Loc2)/{Loc1\==Loc2}
   => {remove(at(Obj,Loc1))}.
```

Although the de larative reading of this rule an be questioned, its pro edural interpretation is lear and useful

> If an obje t is known to be at lo ation $Loc1$ and an assertion is added that it is at some lo ation $Loc2$, distin t from $Loc1$, then the assertion that it is at $Loc1$ should be removed.

The Right Hand Side

The examples seen so far have shown a rules rhs as a single proposition to be "added" to the database. The rhs of a P_{fc} rule has some ri hness as well. The rhs of a rule is a onjun tion of fa ts to be "added" to the database and terms en losed in bra kets whi h represent onditions/a tions whi h are exe uted. As a simple example, onsider the on lusions we might draw upon learning that one person is the mother of another

```
mother(X,Y) =>
  female(X),
  parent(X,Y),
  adult(X).
```

As another example, onsider a rule whi h dete ts bigamists and sends an appropriate warning to the proper authorities

```
spouse(X,Y), spouse(X,Z), {Y\==Z} =>
   bigamist(X),
   {format("~N~w is a bigamist, married
      to both ~w and ~w~n",[X,Y,Z])}.
```

Ea h element in the rhs of a rule is pro essed from left to right — assertions being added to the database with appropriate support and onditions being satisfied. If a ondition an not be satisfied, the rest of the rhs is not pro essed.

We would like to allow rules to be expressed as bi- onditional in so far a possible. Thus, an element in the lhs of a rule should have an appropriate meaning on the rhs as well. What meaning should be assigned to the onditional fa t onstru tion (e.g. P/Q) whi h an o ur in a rules lhs? Su h a term in the rhs of a rule is interpreted as a *conditioned assertion*. Thus the assertion P/Q will mat h a ondition $P\prime$ in the lhs of a rule only if P and $P\prime$ unify and the ondition Q is satisfiable. For example, onsider the rules that says that an obje t being lo ated at one pla e is reason to believe that it is not at any other pla e

```
at(X,L1) => not(at(X,L2))/L2\==L1
```

Note that a *conditioned assertion* is essentially a Horn lause. We would express this fa t in Prolog as the ba kward haining rule

```
not(at(X,L2)) :- at(X,L1),L1\==L2.
```

The differen e is, of ourse, that the addition of su h a onditioned assertion will trigger forward haining whereas the assertion of a new ba kward haining rule will not.

The Truth Maintenance System

As dis ussed in the previous se tion, a forward reasoning system has spe ial needs for some kind of *truth maintenance system*. The P_{fc} system has a rather straightforward TMS system whi h re ords justifi ations for ea h fa t dedu ed by a P_{fc} rule. Whenever a fa t is removed from the database, any justifi ations in whi h it plays a part are also removed. The fa ts that are justified by a removed justifi ation are he ked to see if they are still supported by some other justifi ations. If they are not, then those fa ts are also removed.

Su h a TMS system an be relatively expensive to use and is not needed for many appli ations. Consequently, its use and nature are optional in P_{fc} and are ontrolled by the predi ate $pfcTmsMode/1$. The possible ases are three

- $pfcTmsMode(full)$ - The fa t is removed unless it has *well founded support* (WFS). A fa t has WFS if it is supported by the *user* or by *God* or by a justifi ation all of whose justifi ees have WFS[1].
- $pfcTmsMode(local)$ - The fa t is removed if it has no supporting justifi ations.

[1] Determining if a fact has WFS requires detecting local cycles - see [4] for an introduction

- $pfcTmsMode(none)$ - The fa t is never removed.

A fa t is onsidered to be supported by *God* if it is found in the database with no visible means of support. That is, if P_{fc} dis overs an assertion in the database that an take part in a forward reasoning step, and that assertion is not supported by either the user or a forward dedu tion, then a note is added that the assertion is supported by *God*. This adds additional flexibility in interfa ing systems employing P_{fc} to other Prolog appli ations.

For some appli ations, it is useful to be able to justify a tions performed in the rhs of a rule. To allow this, P_{fc} supports the idea of de laring ertain a tions to be *undoable* and provides the user with a way of spe ifying methods to undo those a tions. Whenever an a tion is exe uted in the rhs of a rule and that a tion is undoable, then a re ord is made of the justifi ation for that a tion. If that justifi ation is later invalidated (e.g. through the retra tion of one of its justifi ees) then the support is he ked for the a tion in the same way as it would be for an assertion. If the a tion does not have support, then P_{fc} trys ea h of the methods it knows to undo the a tion until one of them su eeds.

In fa t, in P_{fc} ,you define an a tion as undoable by ... doing. This
e $pfcUndo/2$. The predi ate $pfcUndo(A1,A2)$ is true if exe uting $A2$ is a possible
on of $A1$. For example, we might want to ouple an assertional representation of a
a graphi al display of them through the use of P_{fc} rules

```
layNode(N,XY)}.
splayArc(N1,N2}.

de(N,XY),eraseNode(N,XY)).
c(N1,N2),eraseArc(N1,N2)).
```

eral limitations, most of whi h it inherits from its Prolog roots. One of the more
P_{fc} rules must be expressible as a set of horn lauses. The pra ti al effe t is that
e a onjun tion of terms whi h are either assertions to be added to the database or
Negated assertions and disjun tions are not permitted, making rules like

```
other(X,Y);father(X,Y)
le(X)
```

hat all variables in a P_{fc} rule have impli it universal quantifi ation. As a result, any
rule whi h remain uninstantiated when the lhs has been fully satisfied retain their
This prevents us from using a rule like

```
nt(Y,Z)
er(X,Z).
```

If we do add this rule and assert *randfather(john,mary)*, then P_{fc} will add the
ons *father(john,* *_)* (i.e. "John is the father of everyone") and *parent(_,mary)* (i.e. "Everyone is Mary's parent").

Another problem asso iated with the use of the Prolog database is that assertions ontaining variables a tually ontain " opies" of the variables. Thus, when the onjun tion

```
add(father(adam,X)), X=able
```

is evaluated, the assertion `father(adam,_G032)` is added to the database, where _G032 is a new variable whi h is distin t from X. As a onsequen e, it is never unified with *able*.

3 Predicates

3.1 Manipulating the Database

add(+P)

The fa t or rule P is added to the database with support oming from the user. If the fa t already exists, an additional entry will not be made (unlike Prolog). If the fa ts already exists with support from the user, then a warning will be printed if $pfcWarnings$ is true. Add/1 always su eeds.

pfc(?P)

The predi ate $pfc/1$ is the proper way to a ess terms in the P_{fc} database. **pfc(P)** su eeds if **P** is a term in the urrent pf database after invoking any ba kward haining rules or is provable by Prolog.

rem(+P)

The first fa t (or rule) unifying with P has its user support removed. $rem/1$ will fail if no there are no P_{fc} added fa ts or rules in the database whi h mat h. If removing the user support from a fa t leaves it unsupported, then it will be removed from the database.

rem2(+P

The first fa t (or rule) unifying with P will be removed from the database even if it has valid justifi ations. $rem/1$ will fail if no there are no P_{fc} added fa ts or rules in the database whi h mat h. If removing the user support from the fa t leaves it unsupported, then it will be removed from the database. If the fa t still has valid justifi ations, then a P_{fc} warning message will be printed and the justifi ations removed.

pfcReset

.

Resets the P_{fc} database by trying to retra t all of the prolog lauses whi h were added by alls to add or by the forward ha91ing me hanism.

Term expansions

P_{fc} defines term expansion pro edures for the operators =¿, ¡= and ¡=¿ so that you an have things like the following in a file to be onsulted

```
foo(X) => bar(X).
=> foo(1).
```

The result will be an expansion to

```
:- add((foo(X) => bar(X)).
:- add(foo(1)).
```

3.2 Control Predicates

This se tion des ribes predi ates to ontrol the forward haining sear h strategy and truth maintenan e operations.

pfcSearch(P)

This predi ate is used to set the sear h strategy that P_{fc} uses in doing forward haing. The argument should be one of dire t,depth,breadth.

pfcTmsMode(Mode)

This predi ate ontrols the method used for truth maintenaan e. The three options are none,lo al, y les. Calling pf TmsMode with an instantiated argument will set the mode to that argument.

- **none** means that no truth maintenan e will be done.
- **local** means that limited truth maintenan e will be done. Spe ifi ally, no y les will be he ked.
- **cycles** means that full truth maintenan e will be done, in luding a he k that all fa ts are well grounded.

pfcHalt

Immediately stop the forward hainging pro ess.

pfcRun

Continue the forward hainging pro ess.

pfcStep

Do one iteration of the forward hainging pro ess.

pfcSelect(P)

Sele t next fa t for forward haining (user defined)

pfcWarnings
pfcNoWarnings

3.3 The TMS

The following predi ates are used to a ess the tms information asso iated with P_{fc} fa ts.

justification(+P,-J)
justification(+P,-Js)

justification(P,J) is true if one of the justifi ations for fa t P is J, where J is a list of P_{fc} fa ts and rules whi h taken together dedu e P. Ba ktra king into this predi ate an produ e additional justifi ations. If the fa t was added by the user, then one of the justifi ations will be the list **[user]**. **justifications(P,Js)** is provided for onvenien e. It binds **Js** to a list of all justifi ations returned by (justifi ation/2).

base(+P,-Ps)

assumptions(+P,-Ps)

pfcChild(+P,-Q)
pfcChildren(+P,-Qs)

pfcDescendant(+P,-Q)
pfcDescendants(+P,-Qs)

3.4 Debugging

pfcTrace
pfcTrace(+Term)
pfcTrace(+Term,+Mode)
pfcTrace(+Term,+Mode,+Condition)

This predi ate auses the addition and/or removal of P_{fc} terms to be tra ed if a spe ified ondition is met. The arguments are as follows

- term - Spe ifies whi h terms will be tra ed. Defaults to _ (i.e. all terms).
- mode - Spe ifies whether the tra ing will be done on the addition (i.e. **add**, removal (i.e. **rem**) or both (i.e. _) of the term. Defaults to _.
- ondition - Spe ifies an additional ondition whi h must be met in order for the term to be tra ed. For example, in order to tra e both the addition and removal of assertions of the age of people just when the age is greater than 100, you an do **pfcTrace(age(_,N),_,N¿100)**.

Thus, alling **pfcTrace** will ause all terms to be tra ed when they are added and removed from the database. When a fa t is added or removed from the database, the lines

```
1
2
```

are displayed, respe tively.

pfcUntrace
pfcUntrace(+Term)
pfcUntrace(+Term,+Mode)
pfcUntrace(+Term,+Mode,+Condition)

The **pfcUntrace** predi ate is used to stop tra ing P_{fc} fa ts. Calling **pfcUntrace(P,M,C)** will stop all tra ing spe ifi ations whi h mat h. The arguments default as des ribed above.

pfcSpy(+Term)
pfcSpy(+Term,+Mode)
pfcSpy(+Term,+Mode,+Condition)

These predi ates set spypoints, of a sort.

pfcQueue

Displays the urrent queue of fa ts in the P_{fc} queue.

showState

Displays the state of Pf , in luding the queue, all triggers, et .

pfcFact(+P)
pfcFacts(+L)

pf Fa t(P) unifies P with a fa t that has been added to the database via P_{fc}.You an ba ktra into it to find more fa ts. pf Fa ts(L) unified L with a list of all of the fa ts asserted by add.

pfcPrintDb
pfcPrintFacts
pfcPrintRules

These predi ates diaply the the entire P_{fc} database (fa ts and rules) or just the fa ts or just the rules.

4 Examples

4.1 Factorial and Fibonacci

These examples show that the P_{fc} ba kward haining fa ility an do su h standard examples as the fa torial and Fibona i fun tions.

Here is a simple example of a P_{fc} ba kward haining rule to ompute the Fibona i series.

```
fib(0,1).
fib(1,1).
fib(N,M) <=
  N1 is N-1,
  N2 is N-2,
  fib(N1,M1),
  fib(N2,M2),
  M is M1+M2.
```

Here is a simple example of a P_{fc} ba kward haining rule to ompute the fa torial fun tion.

```
=> fact(0,1).
fact(N,M) <=
  N1 is N-1,
  fact(N1,M1),
  M is N*M1.
```

4.2 Default Reasoning

This example shows how to define a default rule. Suppose we would like to have a default rule that holds in the absen e of ontradi tory eviden e. We might like to state, for example, that an we should assume that a bird an fly unless we know otherwise. This ould be done as

```
bird(X), ~not(fly(X)) => fly(X).
```

We an, for our onvenien e, define a *default* operator whi h takes a P_{fc} rule and qualifies it to make it a default rule. This an be done as follows

```
default((P => Q)),{pfcAtom(Q)} => (P, ~not(Q) => Q).
```

where **pfcAtom(X)** holds if **X** is a "logi al atom" with respe t to P_{fc} (i.e . not a onjun tion, disjun tion, negation, et).

One we have defined this, we an use it to state that birds fly by default, but penguins do not.

```
% birds fly by default.
=> default((bird(X) => fly(X))).

isa(C1,C2) =>
  % here's one way to do an isa hierarchy.
  {P1 =.. [C1,X],
    P2 =.. [C2,X]},
  (P1 => P2).

=> isa(canary,bird).
=> isa(penguin,bird).

% penguins do not fly.
penguin(X) => not(fly(X)).

% chilly is a penguin.
=> penguin(chilly).

% tweety is a canary.
=> canary(tweety).
```

4.3 KR example

isa hierar hy. roles. types. lassifi ation. et .

4.4 Maintaining Functional Dependencies

One useful thing that P_{fc} an be used for is to automati ally maintain fun tion Dependen ies in the light of a dynami database of fa t. The builtin truth maintenan e system does mu h of this. However, it is often useful to do more. For example, suppose we want to maintain the onstraint that a parti ular obje t an only be lo ated in one pla e at a given time. We might re ord an obje ts lo ation with an assertion **at(Obj,Loc)** whi h states that the urrent lo ation of the obje t **Obj** is the lo ation **Loc**.

Suppose we want to define a P_{fc} rule whi h will be triggered whenever an **at/2** assertion is made and will remove any previous assertion about the same obje t's lo ation. Thus to refle t that an obje t has moved from lo ation A to lo ation B, we need merely add the new information that it is at lo ation B. If we try to do this with the P_{fc} rule

```
at(Obj,Loc1),
at(Obj,Loc2),
{Loc1\==Loc2}
=>
~at(Obj,Loc1).
```

we may or may not get the desired result. This rule will in fa t maintain the onstraint that the database have at most one **at/2** assertion for a given obje t, but whether the one kept is the old or the new depends on the parti ular sear h strategy being used by P_{fc}.In fa t, under the urrent default strategy, the new assertion will be the one retra ted.

We an a hieve the desired result with the following rule

```
at(Obj,NewLoc),
{at(Obj,OldLoc), OldLoc\==NewLoc}
  =>
  ~at(Obj,OldLoc).
```

This rule auses the following behavior. Whenever a new assertion **at(O,L)** is made, a Prolog sear h is made for an assertion that obje t O is lo ated at some other lo ation. If one is found, then it is removed.

We an generalize on this rule to define a meta-predi ate **function(P)** whi h states that the predi ate whose name is **P** represents a fun tion. That is, **P** names a relation of arity two whose first argument is the domain of the fun tion and whose se ond argument is the fun tion's range. Whenever an assertion **P(X,Y)** is made, any old assertions mat hing **P(X,_)** are removed. Here is the P_{fc} rule

```
function(P) =>
  {P1 =.. [P,X,Y],
   P2 =.. [P,X,Z]},
  (P1,{P2,Y\==Z} => ~P2).
```

We an try this with the following results

```
| ?- add(function(age)).
Adding (u) function(age)
Adding age(A,B),{age(A,C),B\==C}=> ~age(A,C)
yes

| ?- add(age(john,30)).
Adding (u) age(john,30)
yes

| ?- add(age(john,31)).
Adding (u) age(john,31)
Removing age(john,30).
yes
```

Of ourse, this will only work for fun tions of exa tly one argument, whi h in Prolog are represented as relations of arity two. We an further generalize to fun tions of any number of arguments (in luding zero), with the following rule

```
function(Name,Arity) =>
  {functor(P1,Name,Arity),
   functor(P2,Name,Arity),
   arg(Arity,P1,PV1),
   arg(Arity,P2,PV2),
   N is Arity-1,
```

```
    merge(P1,P2,N)},
   (P1,{P2,PV1\==PV2} => ~P2).


merge(_,_,N) :- N<1.
merge(T1,T2,N) :-
  N>0,
  arg(N,T1,X),
  arg(N,T2,X),
  N1 is N-1,
  merge(T1,T2,N1).
```

The result is that adding the fa t **function(P,N)** de lares P to be the name of a relation of arity N su h that only the most re ent assertion of the form $P(a_1, a_2, \ldots, a_{n-1}, a_n)$ for a given set of onstants $a_1, \ldots, a_{n-1}$ will be in the database. The following examples show how we might use this to define a predi ate **current_president/1** that identifies the urrent U.S. president and **governor/3** that relates state, a year and the name of its governor.

```
% current_president(Name)
| ?- add(function(current_president,1)).
Adding (u) function(current_president,1)
Adding current_president(A),
       {current_president(B),A\==B}
=>
        ~current_president(B)
yes

| ?- add(current_president(reagan)).
Adding (u) current_president(reagan)
yes

| ?- add(current_president(bush)).
Adding (u) current_president(bush)
Removing current_president(reagan).
yes

% governor(State,Year,Governor)
| ?- add(function(governor,3)).
Adding (u) function(governor,3)
Adding governor(A,B,C),{governor(A,B,D),C\==D}=> ~governor(A,B,D)
yes

| ?- add(governor(pennsylvania,1986,thornburg)).
Adding (u) governor(pennsylvania,1986,thornburg)
yes

| ?- add(governor(pennsylvania,1987,casey)).
Adding (u) governor(pennsylvania,1987,casey)
yes

% oops, we misspelled thornburgh!
| ?- add(governor(pennsylvania,1986,thornburgh)).
Adding (u) governor(pennsylvania,1986,thornburgh)
Removing governor(pennsylvania,1986,thornburg).
yes
```

4.5 Spreadsheets

One ommon kind of onstraints is often found in spreadsheets in whi h one value is determined from a set of other values in whi h the size of the set an vary. This is typi ally found in spread sheets where one ell an be defined as the sum of a olumn of ells. This example shows how this kind of onstraint an be defined in P_{fc} as well. Suppose we have a relation **income/4** whi h re ords a person's in ome for a year by sour e. For example, we might have assertions like

```
income(smith,salary,1989,50000
```

it follows that r is true. However, it is not possible to dire tly en ode this in Prolog so that it an be proven. We an en ode these fa ts in P_{fc} and use a simple proof by ontradi tion strategy embodied in the following Prolog predi ate

```
prove_by_contradiction(P) :- P.
prove_by_contradiction(P) :-
  \+ (not(P) ; P)
  add(not(P)),
    P        -> rem(not(P))
  otherwise -> (rem(not(P)),fail).
```

This pro edure works as follows. In trying to prove P, su eed immediately if P is a know fa t. Otherwise, providing that **not(P)** is not a know fa t, add it as a fa t and see if this gives rise to a proof for (P). if it did, then we have derived a ontradi tion from assuming that **not(P)** is true and **P** must be true. In any ase, remove the temporary assertion **not(P)**.

In order to do the example above, we need to add the following rule or **or** and a rule for general impli ation (en oded using the infix operator **==¿**) whi h generates a regular forward haining rule and its ounterfa tual rule.

```
:- op(1050,xfx,('==>')).

(P ==> Q) =>
  (P => Q),
  (not(Q) => not(P)).

or(P,Q) =>
  (not(P) => Q),
  (not(Q) => P).
```

With this, we an en ode the problem as

```
=> or(p,q).
=> (p ==> x).
=> (q ==> x).
```

When these fa ts are added, the following tra e ensues

```
Adding (u) (A==>B)=>(A=>B), (not(B)=>not(A))
Adding (u) or(A,B)=>(not(A)=>B), (not(B)=>A)
Adding (u) or(p,q)
Adding not(p)=>q
Adding not(q)=>p
Adding (u) p==>x
Adding p=>x
Adding not(x)=>not(p)
Adding (u) q==>x
Adding q=>x
Adding not(x)=>not(q)
```

Then, we an all **prove_by_contradiction/1** to show that **p** must be true

```
| ?- prove_by_contradiction(x).
Adding (u) not(x)
Adding not(p)
Adding q
Adding x
Adding not(q)
Adding p
Removing not(x).
Removing not(p).
Removing q.
Removing not(q).
Removing p.
Removing x.
yes
```

References

[1] M. Genesereth et. al. *MRS Manual.* Te hni al Report, Stanford University, 1983.

[2] Ri h Fritzson and Tim Finin. *Protem - An Inte rated Expert SystemsTool.* Te hni al Report LBS Te hni al Memo Number 84, Unisys Paoli Resear h Center, May 1988.

[3] Drew M Dermott. *DUCK: A Lisp-Based Deductive System.* Te hni al Report, Computer S ien e, Yale University, 1983.

[4] Drew M Dermott and Eugene Charniak. *Introduction to Artificial Intelli ence.* Addison Wesley, 1985.

[5] Nils Nilsson. *Principles of Artificial Intelli ence.* Tioga Publishing Co., Palo Alto, California, 1980.

[6] Charles J. Petrie and Mi hael N. Huhns. *Controllin Forward Rule Inferences.* Te hni al Report ACA-AI-012-88, MCC, January 1988.

```
spouse(X,Y) <=> spouse(Y,X).
spouse(X,Y),gender(X,G1),{otherGender(G1,G2)}
     =>gender(Y,G2).
gender(P,male) <=> male(P).
gender(P,female) <=> female(P).
parent(X,Y),female(X) <=> mother(X,Y).
parent(X,Y),parent(Y,Z) => grandparent(X,Z).
grandparent(X,Y),male(X) <=> grandfather(X,Y).
grandparent(X,Y),female(X) <=> grandmother(X,Y).
mother(Ma,Kid),parent(Kid,GrandKid)
      =>grandmother(Ma,GrandKid).
grandparent(X,Y),female(X) <=> grandmother(X,Y).
parent(X,Y),male(X) <=> father(X,Y).
mother(Ma,X),mother(Ma,Y),{X\==Y}
     =>sibling(X,Y).
```

Figure 1 **Examples of P_{fc} rules which represent common kinship relations**