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 logic programming languages, offers backward chaining as the only reasoning scheme. It is well known that sound and complete reasoning systems can be built using either exclusive backward chaining or exclusive forward chaining [5]. Thus, this is not a theoretical problem. It is also well understood how to "implement" forward reasoning using an exclusively backward chaining system and vice versa. Thus, this need not be a practical problem. In fact, many of the logic-based languages developed for AI applications [3, 1, 6, 2] allow one to build systems with both forward and backward chaining rules.

There are, however, some interesting and important issues which need to be addresses in order to provide the Prolog programmer with a practical, efficient, and well integrated facility for forward chaining. This paper describes such a facility, P_{fc} , which we have implemented in standard Prolog.

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 facts into the regular Prolog database. For example, Figure 1 shows a file of P_{fc} rules and facts which are appropriate for the ubiquitous kinship domain.

The rest of this manual is structured as follows. The next section provides an informal introduction to the P_{fc} language. Section three describes the predicates through which the user calls P_{fc}.The final section gives several longer examples of the use of P_{fc}

Getting and installing P_{fc}

Look for P_{fc} on ftp.cs.umbc.edu in /pub/pfc/.

2 An Informal Introduction to the P_{fc} language

This section describes P_{fc} . We will start by introducing the language informally through a series of examples drawn from the domain of kinship relations. This will be followed by an example and a description of some of the details of its current implementation.

Overview

The P_{fc} package allows one to define forward chaining rules and to add ordinary Prolog assertions into the database in such 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,male) => male(P)
```

This rule states that whenever the fact unifying with $gender(P, male)$ is added to the database, then the fact $male(P)$ is true. If this fact is not already in the database, it will be added. In any case, a record will be made that the validity of the fact $male(P)$ depends, in part, on the validity of this forward chaining rule and the fact which triggered it. To make the example concrete, if we add $gender(john, male)$, then the fact $male(john)$ will be added to the database unless it was already there.

In order to make this work, it is necessary to use the predicate *add/1* rather than *assert/1* in order to assert P_{fc} rules and any facts which might appear in the lhs of a P_{fc} rule.

Compound Rules

A slightly more complex rule is one in which the rule's left hand side is a conjunction or disjunction of conditions:

```
parent(X,Y),female(X) => mother(X,Y)
mother(X,Y);father(X,Y) => parent(X,Y)
```

The first rule has the effect of adding the assertion $mother(X, Y)$ to the database whenever $parent(X, Y)$ and $female(X)$ are simultaneously true for some X and Y. Again, a record will be kept that indicates that any fact $mother(X, Y)$ added by the application of this rule is justified by the rule and the two triggering facts. If any one of these three clauses is removed from the database, then all facts solely dependent on them will also be removed. Similarly, the second example rule derives the parent relationship whenever either the mother relationship or the father relationship is known.

In fact, the lhs of a P_{fc} rule can be an arbitrary conjunction or disjunction of facts. For example, we might have a rule like:

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

P_{fc} handles such a rule by putting it into conjunctive 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-conditional rules, such as:

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

In particular, 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-conditional rules stem from the restrictions that the two derived rules be valid horn clauses. This is discussed in a later section.

Backward-Chaining P_{fc} Rules

P_{fc} includes a special kind of backward chaining rule which is used to generate all possible solutions to a goal that is sought in the process of forward chaining. Suppose we wished to define the *ancestor* relationship as a P_{fc} rule. This could 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 which will never be needed. An alternative is to define the *ancestor* relationship by way of backward chaining rules which are invoked whenever a particular ancestor relationship is needed. In P_{fc} this need arises whenever facts matching the relationship are sought while trying a forward chaining 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 necessary to add some further condition on a rule. Consider a definition of sibling which states:

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

This definition could 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 condition to the lhs of the rule which states the the variables $P1$ and $P2$ must not unify. This is effected by enclosing an arbitrary Prolog goal in braces. When the goals to the left of such a bracketed condition have been fulfilled, then it will be executed. If it can be satisfied, then the rule will remain active, otherwise it will be terminated.

Negation

We sometimes want to draw an inference from the absence of some knowledge. For example, we might wish to encode the default rule that a person is assumed to be male unless we have evidence to the contrary:

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

A lhs term preceded by a $\sim$ is satisfied only if *no* fact in the database unifies with it. Again, the P_{fc} system records a justification for the conclusion which, in this case, states that it depends on the absence of the contradictory evidence. 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 complicated example, consider a rule which states that we should assume that the parents of a person are married unless we know otherwise. Knowing otherwise might consist of either knowing that one of them is married to a yet another person or knowing that they are divorced. We might try to encode 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 conjoined condition

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

does not mean what we want it to mean - that there is no $P3$ distinct 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 qualification `{P3\==P2}` inside the scope of the negation. To achieve this, we introduce the notion of a qualified goal. A lhs term P/C, where P is a positive atomic condition, is true only if there is a database fact unifying with P and condition C is satisfiable. Similarly, a lhs term $\sim P/C$, where P is a positive atomic condition, is true only if there is no database fact unifying with P for which condition C is satisfiable. Our rule can 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 procedural interpretation of a P_{fc} rule is that the conditions in the lhs are checked *from left to right*. One advantage to this is that the programmer can chose an order to the conditions 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 declarative reading of this rule can be questioned, its procedural interpretation is clear and useful:

> If an object is known to be at location $Loc1$ and an assertion is added that it is at some location $Loc2$, distinct 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 richness as well. The rhs of a rule is a conjunction of facts to be "added" to the database and terms enclosed in brackets which represent conditions/actions which are executed. As a simple example, consider the conclusions 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, consider a rule which detects 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])}.
```

Each element in the rhs of a rule is processed from left to right — assertions being added to the database with appropriate support and conditions being satisfied. If a condition can not be satisfied, the rest of the rhs is not processed.

We would like to allow rules to be expressed as bi-conditional 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 conditional fact construction (e.g. P/Q) which can occur in a rules lhs? Such a term in the rhs of a rule is interpreted as a *conditioned assertion*. Thus the assertion P/Q will match a condition $P\prime$ in the lhs of a rule only if P and $P\prime$ unify and the condition Q is satisfiable. For example, consider the rules that says that an object being located at one place is reason to believe that it is not at any other place:

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

Note that a *conditioned assertion* is essentially a Horn clause. We would express this fact in Prolog as the backward chaining rule:

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

The difference is, of course, that the addition of such a conditioned assertion will trigger forward chaining whereas the assertion of a new backward chaining rule will not.

The Truth Maintenance System

As discussed in the previous section, a forward reasoning system has special needs for some kind of *truth maintenance system*. The P_{fc} system has a rather straightforward TMS system which records justifications for each fact deduced by a P_{fc} rule. Whenever a fact is removed from the database, any justifications in which it plays a part are also removed. The facts that are justified by a removed justification are checked to see if they are still supported by some other justifications. If they are not, then those facts are also removed.

Such a TMS system can be relatively expensive to use and is not needed for many applications. Consequently, its use and nature are optional in P_{fc} and are controlled by the predicate $pfcTmsMode/1$. The possible cases are three:

- $pfcTmsMode(full)$ - The fact is removed unless it has *well founded support* (WFS). A fact has WFS if it is supported by the *user* or by *God* or by a justification all of whose justificees have WFS[1].
- $pfcTmsMode(local)$ - The fact is removed if it has no supporting justifications.

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

- $pfcTmsMode(none)$ - The fact is never removed.

A fact is considered to be supported by *God* if it is found in the database with no visible means of support. That is, if P_{fc} discovers an assertion in the database that can take part in a forward reasoning step, and that assertion is not supported by either the user or a forward deduction, then a note is added that the assertion is supported by *God*. This adds additional flexibility in interfacing systems employing P_{fc} to other Prolog applications.

For some applications, it is useful to be able to justify actions performed in the rhs of a rule. To allow this, P_{fc} supports the idea of declaring certain actions to be *undoable* and provides the user with a way of specifying methods to undo those actions. Whenever an action is executed in the rhs of a rule and that action is undoable, then a record is made of the justification for that action. If that justification is later invalidated (e.g. through the retraction of one of its justificees) then the support is checked for the action in the same way as it would be for an assertion. If the action does not have support, then P_{fc} trys each of the methods it knows to undo the action until one of them succeeds.

In fact, in P_{fc} , one declares an action as undoable just by defining a method to accomplish the undoing. This is done via the predicate $pfcUndo/2$. The predicate $pfcUndo(A1, A2)$ is true if executing $A2$ is a possible way to undo the execution of $A1$. For example, we might want to couple an assertional representation of a set of graph nodes with a graphical display of them through the use of P_{fc} rules:

```
at(N,XY) => {displayNode(N,XY)}.
arc(N1,N2) => {displayArc(N1,N2}.

pfcUndo(displayNode(N,XY),eraseNode(N,XY)).
pfcUndo(displayArc(N1,N2),eraseArc(N1,N2)).
```

Limitations

The P_{fc} system has several limitations, most of which it inherits from its Prolog roots. One of the more obvious of these is that P_{fc} rules must be expressible as a set of horn clauses. The practical effect is that the rhs of a rule must be a conjunction of terms which are either assertions to be added to the database or actions to be executed. Negated assertions and disjunctions are not permitted, making rules like

```
parent(X,Y) <=> mother(X,Y);father(X,Y)
male(X) <=> ~female(X)
```

ill-formed.

Another restrictions is that all variables in a P_{fc} rule have implicit universal quantification. As a result, any variables in the rhs of a rule which remain uninstantiated when the lhs has been fully satisfied retain their universal quantification. This prevents us from using a rule like

```
father(X,Y), parent(Y,Z)
    <=> grandfather(X,Z).
```

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

Another problem associated with the use of the Prolog database is that assertions containing variables actually contain "copies" of the variables. Thus, when the conjunction

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

is evaluated, the assertion `father(adam,_G032)` is added to the database, where _G032 is a new variable which is distinct from X. As a consequence, it is never unified with *able*.

3 Predicates

3.1 Manipulating the Database

add(+P)

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

pfc(?P)

The predicate $pfc/1$ is the proper way to access terms in the P_{fc} database. **pfc(P)** succeeds if **P** is a term in the current pfc database after invoking any backward chaining rules or is provable by Prolog.

rem(+P)

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

rem2(+P

The first fact (or rule) unifying with P will be removed from the database even if it has valid justifications. $rem/1$ will fail if no there are no P_{fc} added facts or rules in the database which match. If removing the user support from the fact leaves it unsupported, then it will be removed from the database. If the fact still has valid justifications, then a P_{fc} warning message will be printed and the justifications removed.

pfcReset

.

Resets the P_{fc} database by trying to retract all of the prolog clauses which were added by calls to add or by the forward chaining mechanism.

Term expansions

P_{fc} defines term expansion procedures for the operators =¿, ¡= and ¡=¿ so that you can have things like the following in a file to be consulted

```
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 section describes predicates to control the forward chaining search strategy and truth maintenance operations.

pfcSearch(P)

This predicate is used to set the search strategy that P_{fc} uses in doing forward chaing. The argument should be one of direct,depth,breadth.

pfcTmsMode(Mode)

This predicate controls the method used for truth maintenaance. The three options are none,local,cycles. Calling pfcTmsMode with an instantiated argument will set the mode to that argument.

- **none** means that no truth maintenance will be done.
- **local** means that limited truth maintenance will be done. Specifically, no cycles will be checked.
- **cycles** means that full truth maintenance will be done, including a check that all facts are well grounded.

pfcHalt

Immediately stop the forward chainging process.

pfcRun

Continue the forward chainging process.

pfcStep

Do one iteration of the forward chainging process.

pfcSelect(P)

Select next fact for forward chaining (user defined)

pfcWarnings
pfcNoWarnings

3.3 The TMS

The following predicates are used to access the tms information associated with P_{fc} facts.

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

justification(P,J) is true if one of the justifications for fact P is J, where J is a list of P_{fc} facts and rules which taken together deduce P. Backtracking into this predicate can produce additional justifications. If the fact was added by the user, then one of the justifications will be the list **[user]**. **justifications(P,Js)** is provided for convenience. It binds **Js** to a list of all justifications returned by **(**justification/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 predicate causes the addition and/or removal of P_{fc} terms to be traced if a specified condition is met. The arguments are as follows:

- term - Specifies which terms will be traced. Defaults to _ (i.e. all terms).
- mode - Specifies whether the tracing will be done on the addition (i.e. **add**, removal (i.e. **rem**) or both (i.e. _) of the term. Defaults to _.
- condition - Specifies an additional condition which must be met in order for the term to be traced. For example, in order to trace both the addition and removal of assertions of the age of people just when the age is greater than 100, you can do **pfcTrace(age(_,N),_,N¿100)**.

Thus, calling **pfcTrace** will cause all terms to be traced when they are added and removed from the database. When a fact is added or removed from the database, the lines

```
1
2
```

are displayed, respectively.

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

The **pfcUntrace** predicate is used to stop tracing P_{fc} facts. Calling **pfcUntrace(P,M,C)** will stop all tracing specifications which match. The arguments default as described above.

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

These predicates set spypoints, of a sort.

pfcQueue

Displays the current queue of facts in the P_{fc} queue.

showState

Displays the state of Pfc, including the queue, all triggers, etc.

pfcFact(+P)
pfcFacts(+L)

pfcFact(P) unifies P with a fact that has been added to the database via P_{fc}You can backtrac into it to find more facts. pfcFacts(L) unified L with a list of all of the facts asserted by add.

pfcPrintDb
pfcPrintFacts
pfcPrintRules

These predicates diaply the the entire P_{fc} database (facts and rules) or just the facts or just the rules.

4 Examples

4.1 Factorial and Fibonacci

These examples show that the P_{fc} backward chaining facility can do such standard examples as the factorial and Fibonacci functions.

Here is a simple example of a P_{fc} backward chaining rule to compute the Fibonacci 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} backward chaining rule to compute the factorial function.

```
=> 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 absence of contradictory evidence. We might like to state, for example, that an we should assume that a bird can fly unless we know otherwise. This could be done as:

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

We can, for our convenience, define a *default* operator which takes a P_{fc} rule and qualifies it to make it a default rule. This can be done as follows:

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

where **pfcAtom(X)** holds if **X** is a "logical atom" with respect to P_{fc} (i.e . not a conjunction, disjunction, negation, etc).

One we have defined this, we can 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 hierarchy. roles. types. classification. etc.

4.4 Maintaining Functional Dependencies

One useful thing that P_{fc} can be used for is to automatically maintain function Dependencies in the light of a dynamic database of fact. The builtin truth maintenance system does much of this. However, it is often useful to do more. For example, suppose we want to maintain the constraint that a particular object can only be located in one place at a given time. We might record an objects location with an assertion **at(Obj,Loc)** which states that the current location of the object **Obj** is the location **Loc**.

Suppose we want to define a P_{fc} rule which will be triggered whenever an **at/2** assertion is made and will remove any previous assertion about the same object's location. Thus to reflect that an object has moved from location A to location B, we need merely add the new information that it is at location 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 fact maintain the constraint that the database have at most one **at/2** assertion for a given object, but whether the one kept is the old or the new depends on the particular search strategy being used by P_{fc}. In fact, under the current default strategy, the new assertion will be the one retracted.

We can achieve the desired result with the following rule:

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

This rule causes the following behavior. Whenever a new assertion **at(O,L)** is made, a Prolog search is made for an assertion that object O is located at some other location. If one is found, then it is removed.

We can generalize on this rule to define a meta-predicate **function(P)** which states that the predicate whose name is **P** represents a function. That is, **P** names a relation of arity two whose first argument is the domain of the function and whose second argument is the function's range. Whenever an assertion **P(X,Y)** is made, any old assertions matching **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 can 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 course, this will only work for functions of exactly one argument, which in Prolog are represented as relations of arity two. We can further generalize to functions of any number of arguments (including 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 fact **function(P,N)** declares P to be the name of a relation of arity N such that only the most recent assertion of the form $P(a_1, a_2, \ldots, a_{n-1}, a_n)$ for a given set of constants $a_1, \ldots, a_{n-1}$ will be in the database. The following examples show how we might use this to define a predicate **current_president/1** that identifies the current 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 common kind of constraints is often found in spreadsheets in which one value is determined from a set of other values in which the size of the set can vary. This is typically found in spread sheets where one cell can be defined as the sum of a column of cells. This example shows how this kind of constraint can be defined in P_{fc} as well. Suppose we have a relation **income/4** which records a person's income for a year by source. For example, we might have assertions like:

```
income(smith,salary,1989,50000).
income(smith,interest,1989,500).
income(smith,dividends,1989,1200).
income(smith,consulting,1989,2000).
```

We might also with to have a relation **total_income/3** which records a person's total income for each year. Given the database above, this should be:

```
total_income(smith,1989,53700).
```

One way to do this in P_{fc} is as follows:

```
income(Person,Source,Year,Dollars) => {increment_income(Person,Year,Dollars)}.

=> pfcUndoMethod(increment_income(P,Y,D),decrement_income(P,Y,D)).

increment_income(P,Y,D) :-
  (retract(total_income(P,Y,Old)) -> New is Old+D ; New = D),
  assert(total_income(P,Y,New)).

decrement_income(P,Y,D) :-
  retract(total_income(P,Y,Old)),
  New is Old-D,
  assert(total_income(P,Y,New)).
```

We would probably want to use the P_{fc} rule for maintaining functional Dependencies described in Section 4.4 as well, adding the rule:

```
=> function(income,4).
```

4.6 Extended Reasoning Capability

The truth maintenance system in P_{fc} makes it possible to do some reasoning that Prolog does not allow. From the facts

$$p \lor q$$
$$p \rightarrow r$$
$$q \rightarrow r$$

it follows that r is true. However, it is not possible to directly encode this in Prolog so that it can be proven. We can encode these facts in P_{fc} and use a simple proof by contradiction strategy embodied in the following Prolog predicate:

```
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 procedure works as follows. In trying to prove P, succeed immediately if P is a know fact. Otherwise, providing that **not(P)** is not a know fact, add it as a fact and see if this gives rise to a proof for **(**P). if it did, then we have derived a contradiction from assuming that **not(P)** is true and **P** must be true. In any case, 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 implication (encoded using the infix operator **==¿**) which generates a regular forward chaining rule and its counterfactual rule.

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

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

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

With this, we can encode the problem as:

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

When these facts are added, the following trace 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 can call **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.* Technical Report, Stanford University, 1983.

[2] Rich Fritzson and Tim Finin. *Protem - An Integrated Expert SystemsTool.* Technical Report LBS Technical Memo Number 84, Unisys Paoli Research Center, May 1988.

[3] Drew McDermott. *DUCK: A Lisp-Based Deductive System.* Technical Report, Computer Science, Yale University, 1983.

[4] Drew McDermott and Eugene Charniak. *Introduction to Artificial Intelligence.* Addison Wesley, 1985.

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

[6] Charles J. Petrie and Michael N. Huhns. *Controlling Forward Rule Inferences.* Technical 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**