Proof Techniques, an Extended Example

Here is an example from relational calculus to illustrate each of the four methods of proof (case-analysis or truth tables, natural deduction, resolution, and boundary logic) The example can be viewed as a knowledge-base query. A ***knowledge-base*** (KB) is a collection of ***facts*** (which contain no variables) and ***rules*** (which contain variables, and are usually stated in an if...then... format). Both the fact-base and the rule-base have been greatly simplified for this example. The deductive processes are essentially the same, regardless of the complexity of the knowledge-base.

A Relational Calculus (Database Query) Example

Facts:

```
F1.   (George  is-the-father-of  Harry)
F2.   (Rita    is-the-sister-of  Harry)
F3.   (Rita    is-never-married)
F4.   (Harry   is-a-male)
```

Rules:

```
R1.   If  (_1_  is-the-father-of  _3_)
          and (_2_  is-the-sister-of  _3_)
              then (_1_  is-the-father-of  _2_)

R2.   If  (_4_  is-the-father-of  _5_)
          and (_5_  is-a-male)
              then (_5_  is-the-son-of  _4_)

R3.   If  (_6_  is-the-son-of  _7_)
            then (_6_  has-same-last-name-as  _7_)

R4.   If  (_8_  is-the-father-of  _9_)
          and (_9_  is-never-married)
              then (_9_  has-same-last-name-as  _8_)

R5.   If  (_10_  has-same-last-name-as  _11_)
          and (_10_  has-same-last-name-as  _12_)
              then (_11_  has-same-last-name-as  _12_)

R6.   If  (_13_ has-same-last-name _14_)
            then  (_14_ has-same-last-name _13_)
```

Query:

```
Q.         (Harry has-same-last-name-as  Rita)
```

Abbreviations:

```
George                              =     g
Rita                                =     r
Harry                               =     h
(_1_  is-the-father-of  _2_)        =     1F2
(_1_  is-the-sister-of  _2_)        =     1T2
(_1_  is-the-son-of  _2_)           =     1S2
(_1_  has-same-last-name-as  _2_)   =     1L2
```

```
(_1_  is-a-male)                    =     1M
(_1_ is-never-married)              =     1N

Variables will be integers      =   {1, 2, 3, ...}
```

Abbreviated knowledge-base:

```
F1.   gFh
F2.   rTh
F3.   rN
F4.   hM

R1.   if   1F3   and    2T3   then   1F2
R2.   if   4F5   and    5M    then   5S4
R3.   if   6S7   then   6L7
R4.   if   8F9   and    9N    then   9L8
R5.   if  10L11  and   10L12  then  11L12       transitivity
R6.   if  13L14  then  14L13                    commutativity

Q0.   hLr
```

This particular example was designed with these objectives in mind:

1. Intuitive semantics, easy for a human to understand
2. Tractable size but enough to illustrate both natural and algorithmic processes
3. Simple but non-trivial proof in natural deduction
4. Easy forward-chaining proof in case-analysis (as a consequence of this, a complex backward chaining proof)
5. Surprising proof in algorithmic resolution
6. Illustrative proof of minimal boundary techniques, including more complex set techniques.
7. Difficulty general transitivity and commutativity rules
8. Tricky and subtle knowledge engineering issues.

Note: in pattern-matching systems, there is no substantive difference between algebraic functions (ie. functions which are not evaluated) and relations.

Natural Deduction

The natural deduction approach is to ***use reason*** to show that Harry and Rita have the same last name because George is their common father and Rita has never married. We show that George is the father of both Harry and Rita, then we show that George has the same last name as both Harry and Rita, then we conclude that Harry and Rita have the same last name. Although the logic is clear, the syntactic transformations to get the rules to confirm the logic require the additional skill of pattern-matching through unification.

```
Show  gFh          F1.   gFh           given
Show  gFr          F2.   rTh
                   R1.   gFh and rTh   therefore  gFr
Show  gLh          R3.   gFh and hM    therefore  hSg
```

```
                  R3.   hSg          therefore  hLg
                  R6.   hLg          therefore  gLh
Show  gLr         R4.   gFr and rN   therefore  rLg
                  R6.   rLg          therefore  gLr
Show  hLr         R5.   gLh and gLr  therefore  hLr
```

Case-analysis and Chaining

Truth tables list all possible facts. In a KB, rules can be seen as sets of facts that have yet to be enumerated. The identifying characteristic of case-analysis is that no variables are included in the final form of rules or queries. One approach is to substitute all possible variable bindings into the rules in all possible combinations. Since we have three people {George, Rita, Harry} in the KB, and all variables refer to these three people, each variable has 3 cases, and each rule would have 3^n cases, where n is the number of different variables in the rule. In the example, this would generate (3^3 + 3^2 + 3^2 + 3^2 + 3^3 + 3^2) = 90 rule cases for the six rules.

A more efficient procedure would be to use the known facts to constrain the generation of cases. We begin with the known facts and then use the rules to (indiscriminately) generate all the other possible facts that are consistent with both the initial facts and the rules. The forward generation of facts from initial conditions and rules is called ***forward-chaining***. We attempt to unify each fact with the premise of each rule; when unification is successful, the conclusion of the rule is asserted as a new fact.

The order of enumeration of facts (the ***enumeration strategy***) is a significant issue. The order in which facts are applied to rules determines which new facts get enumerated first. Since new facts themselves may trigger applications of rules, a choice can be made between ***depth-first*** enumeration (following new facts first) or ***breadth-first*** enumeration (following old facts first). Often a single fact may unify with one premise of a rule which requires two or more facts to fulfill its premise. In this case, a new, shorter rule is asserted.

Below we use the following strategy: first all new facts are generated, then they are in turn used to generate more facts. Duplications has been suppressed (this is called an ***occurs-check***). Using the current facts to generate more facts is called a ***set-of-support*** strategy, since the set of known facts support the conclusions.

```
F5.   F1+F2, R1:      gFr
F6.   F1+F4, R2:      hSg      no other rules unify, use new facts
F7.   F5+F3, R4:      rLh
F8.   F6,    R3:      hLg
F9.   F7,    R6:      hLr      QED
```

Note that this algorithmic proof is shorter than the natural deduction proof. It is still not optimal, since step F8 was unnecessary. Algorithmic proof is always committed to following a blind strategy, trading thought and efficiency for ease of implementation. There is a general computational heuristic here: almost always it is better to implement blind brute force rather than subtle computational intelligence. The corollary to this heuristic is that brute force only works with the appropriate data structure. It is almost always better to apply design intelligence to the representation of a problem than to the algorithm.

Another strategy is to use the query to generate all possible queries stemming backwards from the target query, until the existing facts terminate the search. Queries are matched with the conclusions of rules; the premises of these rules are then the new queries. This technique is called ***backward-chaining***. We first generate queries which can be answered by single facts, then queries which require more than one fact, finally trying to bind queries which contain variables to the initial fact base. The example follows:

```
Q0.                          hLr    ?
Q1.   Q0,  R3:               hSr    ?
Q2.   Q0,  R6:               rLh    ?
Q3.   Q2,  R3:               rSh    ?     No other simple queries
Q4.   Q0,  R4:     rFh  and  hN     ?
Q5.   Q0,  R5:    20Lh  and 20Lr    ?     Introduce new variable numbers
Q6.   Q1,  R2:     rFh  and  hM     ?     =>  rFh    using F4
Q7.   Q2,  R5:    21Lr  and 21Lh    ?     duplicate of Q5
Q8.   Q3,  R2:     hFr  and  rM     ?
Q9.   Q6,  R1:     rF22 and  hT22   ?     No more bindings or ground Qs
Q10.  Q5a, R3:              23Sh    ?     Begin using variable Qs
Q11.  Q5b, R3:              24Sr    ?
Q12.  Q10, R2:     hF25 and 25M     ?
Q13.  Q11, R2:     rF26 and 26M     ?
Q14.  Q5a, R6:               hL27   ?
Q15.  Q5b, R6:               rL28   ?
Q16.  Q14, R3:               hS29   ?
Q17.  Q15, R3:               rS30   ?
Q18.  Q16, R2:    31Fh  and  hM     ?     => 31Fh    using F4
Q19.  Q17, R2:    32Fr  and  rM     ?
Q20.  Q18, F1:               gFh    ?     bind 31 to g using fact F1
```

At this point we have back-chained to an initial fact, `gFh`. Reversing the logic, this fact combined with F4 and R2 (see line Q18) answer line Q16, binding variable 29 to `g`. This answers Q14, binding 27 to `g`. Q14 answers the first part of Q5 (that is Q5a), binding 20 to `g`, and leaving the query sequence below as Q5b. (While we are at an interrupt, note that if the below sequence fails, the queries would pick up where they left off, at Q12 where 25 would bind to `h` using F4. This would create the query `hFh` ? and so on)

```
Q21.  Q5b:                   gLr    ?
Q22.  Q21, R3:               gSr    ?
Q23.  Q21, R6:               rLg    ?
Q24.  Q23, R3:               rSg    ?     No other simple queries
Q25.  Q21, R4:     rLg  and  gN     ?
Q26.  Q21, R5:    33Lg  and 33Lr    ?
Q27.  Q22, R2:     rFg  and  gM     ?
Q28.  Q23, R4:     gFr  and  rN     ?     =>  gFr    using F3
Q29.  Q23, R5:    34Lr  and 34Lg    ?
Q30.  Q28, R1:     gF35 and  rT35   ?     No other grounded queries
Q31.  Q30, F1:     gFh  and  rTh    ?     Bind 35 to h using F1
Q32.  Q31, F2:               rTh    ?     =>  True   using F2
```

We have now reached a final conclusion, since all queries have been answered. Reconstructing the path in reverse order:

```
Q32:        rTh
Q31:        Q32 and gFh
Q30:        Q31
            R1              thus gFr
Q28:        Q30 and rN
            R4              thus rLg
Q23:        Q28
            R6              thus gLr
Q21:        Q23

Q20:        gFh
Q18:        Q20 and hM
            R2              thus hSg
Q16:        Q18
            R3              thus hLg
Q14:        Q16
            R6              thus gLh

Q5:         Q14 and Q21
            R5              thus hLr
Q0:         Q5              QED
```

Note that this proof is similar to the natural deduction, and not as direct as the forward-chaining proof. These differences are an artifact of the particular KB, and are not general. Some KBs are particularly efficient for forward-chaining and some are particularly efficient for backward-chaining. In general, which method is best depends on the specific query, on the particular KB, and on the way in which each rule is formulated (see the **Addendum**). Usually the methods need to be mixed. The resolution technique accomplishes this mixing.

Resolution

In resolution, the KB is converted into sets of clauses. A ***clause*** is a set of both positive or negative atoms joined by disjunction. A KB is a set of clauses. New clauses are added by matching and deleting positive and negative atoms which unify across two clauses. For instance, the logical form `(if A then B)` is converted into the equivalent form `((not A) or B)`, which then is turned into a set of atoms {~A, B}. Resolution looks like this:

{~C, A} resolve-with {~A, B} ==> add {~C, B}

This can be read for logic as `((C implies A) and (A implies B) therefore (C implies B))`. Since resolution is an algorithm, we proceed down the list of clauses in a linear fashion. The query is negated, and we hope to resolve it with an assertion of the positive fact to end the resolution with an empty clause. This looks like:

{A} resolve-with {~A} ==> { }

Several resolution strategies are possible, based on the structure of each clause. For instance facts (clauses with single positive atoms) could be resolved first. Or clauses with single atoms regardless of polarity could be resolved first. Another strategy might be to resolve all instance of a particular relation first. The strategy used below is to resolve all singular clauses first.

```
F1.   {gFh}
F2.   {rTh}
F3.   {rN}
F4.   {hM}
Q.    {~hLr}

R1.   {~1F3, ~2T3, 1F2}            if  1F3  and   2T3  then  1F2
R2.   {~4F5, ~5M, 5S4}             if  4F5  and   5M   then  5S4
R3.   {~6S7, 6L7}                  if  6S7            then  6L7
R4.   {~8F9,  ~9N, 9L8}            if  8F9  and   9N   then  9L8
R5.   {~10L11, ~10L12, 11L12}      if 10L11 and  10L12 then 11L12
R6.   {~13L14, 14L13}              if 13L14            then 14L13

C1.   {~20Th, gF20}          F1,  R1     rename varaibles
C2.   {~hM, hSg}             F1,  R2
C3.   {~hN, hLg}             F1,  R4
C4.   {~21Fh, 21Fr}          F2,  R1
C5.   {~22Fr, rL22}          F3,  R4
C6.   {~23Fh, hS23}          F4,  R2
C7.   {~hSr}                 Q,   R3
C8.   {~rFh, ~hN}            Q,   R4
C9.   {~24Lh, ~24Lr}         Q,   R5
C10.  {~rLh}                 Q,   R6
C11.  {gFr}                  F1,  C4
C12.  {hSg}                  F1,  C6
C13X  {gFr}                  F2,  C1     duplicate of C11
C14X  {hSg}                  F4,  C2     duplicate of C12

C15.  {~rFh, ~hM}            C7,  R2     begin using new facts
C16.  {~rFh}                 C7,  C6
C17.  {~rSh}                 C10, R3
C18.  {~hFr, ~rN}            C10, R4
C19X  {~25Lr, ~25Lh}         C10, R5     duplicate of C9
C20X  {~hLr}                 C10, R6     duplicate of query
C21.  {~hFr}                 C10, C5
C22.  {~26Tr, gF26}          C11, R1
C23.  {~rM, rSg}             C11, R2
C24.  {~rN, rLg}             C11, R4
C25.  {rLg}                  C11, C5
C26.  {hLg}                  C12, R3
C27X  {~hFr}                 F3,  C18    duplicate of C21
C28X  {rLg}                  F3,  C24    duplicate of C27
C29X  {~rFh}                 F4,  C15    duplicate of C16

C30.  {~rF27, ~hT27}         C16, R1     begin with new facts again
C31.  {~hFr, ~rM}            C17, R2
C32.  {~hF28, ~rT28}         C21, R1
C33.  {~hFh}                 C21, C4
C34.  {~rL29, gL29}          C25, R5
C35.  {~rL30, 30Lg}          C25, R5
C36.  {gLr}                  C25, R6
C37.  {~hL31, gL31}          C26, R5
C38.  {~hL32, 32Lg}          C26, R5
```

```
C39.  {gLh}                  C26, R6
C40X  {~hFh}                 F2,  C32    duplicate of C33
C41.  {gLg}                  C25, C34
C42X  {gLg}                  C25, C35    duplicate of C41
C43X  {gLg}                  C26, C37    duplicate of C41
C44X  {gLg}                  C26, C38    duplicate of C41
C45.  {~hFh, ~hTh}           C33, R1     begin with new facts again
C46.  {~gL33, rL33}          C36, R5
C47.  {~gL34, 34Lr}          C36, R5
C48X  {rLg}                  C36, R6     duplicate of C27
C49.  {~gLh}                 C36, C9     resolves with C39 to {}
C50.  {~gL35, hL35}          C39, R5
C51.  {~gL36, 36Lh}          C39, R5
C52X  {hLg}                  C39, R6     duplicate of C25
C53.  {~gLr}                 C39, C9     resolves with C36 to {}
C54.  {~gL37, gL37}          C41, R5
C55.  {}                     C54         QED.
```

The proof terminated with a clause which has a negative and a positive instance of the same atom. There are many observations to be made in this example. Let's begin by unwinding the logic of the proof. When the non-productive resolutions are pruned, the proof is quite straight forward and short.

```
C54:  {~gL37, gL37}
  R5:       {~10L11, ~10L12, 11L12}
  C41:      {gLg}
    C34:          {~rL29, gL29}
      C25:              {rLg}
                            proof below
      R5:               {~10L11, ~10L12, 11L12}
    C25:          {rLg}
      C5:               {~22Fr, rL22}
        F3:                 {rN}
        R4:                 {~8F9,  ~9N, 9L8}
      C11:              {gFr}
        F1:                 {gFh}
        C4:                 {~21Fh, 21Fr}
          F2:                   {rTh}
          R1:                   {~1F3, ~2T3, 1F2}
```

First, the resolution proof adopted a non-intuitive strategy, arguing from absurdity that a person cannot both have the same last name as someone (variable 37 in C54) and not have the same last name as that someone. This approach does not rely on any semantic knowledge about last names, obviously the computation does not understand naming conventions. The consequence is built into the transitivity rule (R5) itself.

Note the recursive use of C25. The established fact `rLg` (from C25) is used with R5 to construct the smaller rule `(if rL29 then gL29)`, if Rita has the same last name as someone, then so does George. It is then used again with that rule (C25 + C34) to show that the unknown person is George himself! Finally R5 is used again with the fact that George has his own last name to terminate the proof. Non-intuitive proofs and proof strategies are characteristic of algorithmic proof systems.

The proof would have been substantively different if R6, the commutative rule for last-names had not been included. In fact, it is not necessary for a proof. In this resolution proof, it is surprising that R2, R3, R6, and F4 were not used at all, even though from a natural deduction perspective they appear mandatory.

Note also the many convergent proofs toward the end. Had C54 not occurred, both C49 and C53 would have terminated the proof during the next cycle. Again, multiple paths with high redundancy are characteristic of algorithmic techniques.

Note also that the distinction between forward and backward chaining is largely lost, since matching positive facts and negative facts uses the same algorithm without distinction. The algorithmic proof followed all paths at the same time, taking small steps along each possible path without regard to conclusions or duplications.

Other control strategies for the resolution would have resulted in different proofs and even different proof strategies. It may have been more efficient, for example, to resolve the new facts with the shorter new rules first, before using the original rules, since the original rules R1 and R5 introduced excess variables.

In resolution, it is possible to resolve rules together, as well as just to follow facts. For example:

```
R2.   {~4F5, ~5M, 5S4}
R3.   {~6S7, 6L7}              ==>    {~20F21, ~21M, 21L20}
```

This generates a new rule, which is more direct for the purposes of the question that has been asked. When to do this becomes clear in the following boundary logic approach.

Boundary Logic

Again we transcribe the rules into a new, boundary, notation:

```
R1:   ( ((1F3) (2T3)) ) 1F2              if  1F3  and  2T3  then  1F2
R2:   ( ((4F5) (5M)) ) 5S4               if  4F5  and  5M   then  5S4
R3:   (6S7) 6L7                          if  6S7            then  6L7
R4:   ( ((8F9) (9N)) ) 9L8               if  8F9  and  9N   then  9L8
R5:   ( ((10L11) (10L12)) ) 11L12        if 10L11 and 10L12 then 11L12
R6:   (13L14)  14L13                     if 13L14           then 14L13
```

In this notation, some redundant logical structure can be seen at the level of individual rules. We simplify the rules individually using Involution:

```
R1:   (1F3) (2T3) 1F2
R2:   (4F5) (5M) 5S4
R3:   (6S7) 6L7
R4:   (8F9) (9N) 9L8
R5:   (10L11) (10L12) 11L12
R6:   (13L14)  14L13
```

The boundary approach is based on reducing the entire collection of rules and facts as a whole. Rather than accumulate new facts, all the facts are combined into a single "conjunction of facts and rules implies conclusion" form. The general template is:

```
( ((fact1) ... (factn) (rule1)... (rulen)) ) query
```

which simplifies to

```
   (fact1) ... (factn) (rule1)... (rulen)    query
```

For the example, the template is

```
      (F1) (F2) (F3) (F4) (R1) (R2) (R3) (R4) (R5) (R6) Q
```

and the specific structure is

```
(gFh) (rTh) (rN) (hM)              facts
((1F3)(2T3) 1F2)                   R1
((4F5)(5M) 5S4)                    R2
((6S7) 6L7)                        R3
((8F9)(9N) 9L8)                    R4
((10L11)(10L12) 11L12)             R5
((13L14)  14L13)                   R6
hLr                                query
```

The boundary approach has taken yet another step away from intuition, now rules and facts are no longer distinguished. Like resolution, there is only one primary transformation, Pervasion. The idea is use the forms on the outside to extract their matching forms from the inside. Again, the matching technique is unification. Unlike resolution, the primary boundary transformation of Pervasion is augmented with two other transformations. Involution cleans up irrelevant logical distinctions, and Dominion tells the process when to stop.

Rule simplification and compilation

It is a good idea to simplify rules first, since they are abstractions applying to all facts, and are the source of complexity.

The first observation is that transitivity (R5) and commutativity (R6) apply all the time. They are not specific enough to help with deductions, but they do help to broaden the generality of facts. Use these rules only to generate new facts, not as part of a deduction.

Since the S relation shows up only once as a premise (in R3) and once as a conclusion (in R2), it can be compiled away. There is only one way to use (that is, to instantiate) the S relation, going from the premises of R2 to the conclusion of R3. In general we do not want to lose the ability to use either R2 or R3 by themselves (for instance in the case that the query is about an S relation), so we compile the S relation dynamically, in the presence of a known query.

Compile rules R2 and R3 into R23, using resolution `(A B) ((B) C) ==> (A C)`

```
S:  6 => 5,  7 => 4
((4F5)(5M) 5S4)  ((6S7) 6L7)  ==>   ((4F5)(5M) 5L4)
```

The new knowledge base:

```
(gFh) (rTh) (rN) (hM) hLr
((1F3)(2T3) 1F2) ((4F5)(5M) 5L4) ((8F9)(9N) 9L8)
((10L11)(10L12) 11L12) ((13L14) 14L13)
```

Should a rule have more than one premise, the ability to branch using the simple rule is lost in compiling. So, for instance, it is not possible to compile the `F` relation, even though it shows up only once as a conclusion (in R1).

Forced bindings

Now we make all forced bindings between the facts and the remaining rules. The Pervasion transformation says that a form on the outside of a boundary must match a form on the inside of a boundary, using unification as the matching technique. When a match is found, bind the variables and extract the inner form:

```
xAy  (1A2  1B3)  ==>  xAy  (xB3)
```

In the example KB, we have the fact `(rN)` on the outside which matches the inner form of R4 `((6F8) (9N) 9L8)`, binding the variable 9 to the atom `r` and erasing the `(9N)`:

```
(rN) ((8F9)(9N) 9L8)  =>  (rN) ((8Fr) rL8)
```

There is only one `(rN)` form on the outside and only one in all the insides, so there is only one possible extraction of an `N` relation. In this example, extracting `(rN)` leaves the knowledge base looking like:

```
N:  9 => r              (rN) ((8F9)(9N) 9L8)  =>  (rN) ((8Fr) rL8)
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr
((1F3)(2T3) 1F2) ((4F5)(5M) 5L4) ((8Fr) rL8)
((10L11)(10L12) 11L12) ((13L14) 14L13)
```

In general, there will be more than one fact matching each inner form, and more than one binding for each variable. This is what makes query management hard. The boundary approach lets us bind all possible variables, using sets of facts rather than individual facts. The set-based boundary approach would extract all matches, binding the variable to a *set* of matches.

We continue the forced (only one choice) bindings, using the strategy of binding the least number of variables first (ie using facts to their full extent). Portions of rules can be deleted when there is no possible way of using them again.

```
M:  5 => h              (hM) ((4F5)(5M) 5L4)  =>  (hM) ((4Fh) hL4)
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr
((1F3)(2T3) 1F2) ((4Fh) hL4) ((8Fr) rL8)
((10L11)(10L12) 11L12) ((13L14) 14L13)
```

There are several different strategies now available, the worst of which is to use either of the general transitivity or commutativity rules. `(gFh)` could extract `(4Fh)`, but this is premature since the rule portion could not be deleted in the presence of other `F` relations in the KB (in particular R1 may generate a need for the remaining portion of R23). Again seeking uniqueness and specificity, the best approach is to select the `T` relation which has only single occurrences on the outside and the inside of the KB.

```
T:  2 => r,  3 => h     (rTh) ((1F3)(2T3) 1F2)  =>  (rTh) ((1Fh) 1Fr)
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr
((1Fh) 1Fr) ((4Fh) hL4) ((8Fr) rL8)
((10L11)(10L12) 11L12) ((13L14) 14L13)
```

Now the `F` relation in both R1 and R23 should be extracted. Neither R1 nor R23 can be deleted. Along the way, a third possible `F` extract is generated and taken.

```
F:  1 => g              (gFh) ((1Fh) 1Fr)  =>  (gFh) (gFr)
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr
(gFr) ((4Fh) hL4) ((8Fr) rL8) ((1Fh) 1Fr)
((10L11)(10L12) 11L12) ((13L14) 14L13)
```

```
F:  4 => g              (gFh) ((4Fh) hL4)  =>  (gFh) (hLg)
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr
(gFr) (hLg) ((8Fr) rL8) ((4Fh) hL4) ((1Fh) 1Fr)
((10L11)(10L12) 11L12) ((13L14) 14L13)
```

```
F:  8 => g              (gFr) ((8Fr) rL8)  =>  (gFr) (rLg)
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr
(gFr) (hLg) (rLg) ((8Fr) rL8) ((4Fh) hL4) ((1Fh) 1Fr)
((10L11)(10L12) 11L12) ((13L14) 14L13)
```

There remains only one path for implication of F relations, that is the backward binding of `hLr` to the remains of R23. By taking this step, we are then free to erase all `F` rules.

```
F:  4 => r              hLr ((4Fh) hL4)  =>  hLr ((rFh))  =>  rFh
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr (gFr) (hLg) (rLg) rFh
((10L11)(10L12) 11L12) ((13L14) 14L13)
```

We have used the "query" `hLr` to generate another query `rFh`. There are now no more forced bindings, so we must use transitivity or commutativity of `L` to generate new facts. Finally we must use the branching rules, but with the comfort that every step thus far was without choice. Note that we can now focus on generating only new `L` facts.

The pair of facts `(hLg)(rLg)` provide a set match for either of the transitivity premises, but there is still a minimal approach to be taken. The commutativity rule has only one match, and further the `hLr` form only matches one form within the commutativity rule. We make that binding:

```
L:  14 => h, 13 => r    hLr ((13L14) 14L13)  =>  hLr ((rLh))  =>  rLh
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr (gFr) (hLg) (rLg) rFh
((10L11)(10L12) 11L12) ((13L14) 14L13) rLh
```

Above, we used the query `hLr` to extract a conclusion from R6. The resulting form which is not inside a boundary is also a query; that is, we would know `hLr` if we could show `rLh`. The critical point here is that we cannot collapse the commutativity rule out of the KB because there are facts present which could use it again. R6 can be used in either direction, forward or backward. The appropriate strategy now is to go ahead and make full use of R6 in the forward direction, with the set of bindings from `(hLg)` and `(rLg)`:

```
L:  13 => h, 14 => g    (hLg) ((13L14) 14L13)  => (hLg) (gLh)
L:  13 => r, 14 => g    (rLg) ((13L14) 14L13)  => (rLg) (gLr)
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr (gFr) (hLg) (rLg) rFh
((10L11)(10L12) 11L12) (gLh) (gLr) rLh  ((13L14) 14L13)
```

We now face many branches, but we have constrained them to only one rule, R5. There are two uses of transitivity in the backward direction, reasoning from queries, so we bind them both, without eliminating the rule.

```
L:  11 => h, 12 => r    hLr ((10L11)(10L12) 11L12) => hLr ((10Lh)(10Lr))
L:  11 => r, 12 => h    rLh ((10L11)(10L12) 11L12) => rLh ((10Lr)(10Lh))
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr (gFr) (hLg) (rLg) rFh
((10L11)(10L12) 11L12) (gLh) (gLr) rLh
((13L14) 14L13) ((10Lh)(10Lr)) ((10Lr)(10Lh))

L:  10 => g             (gLh) ((10Lh)(10Lr))  =>  (gLh) ((gLr))  =>  gLr
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr (gFr) (hLg) (rLg) rFh
((10L11)(10L12) 11L12) (gLh) (gLr) rLh
((13L14) 14L13) gLr ((10Lr)(10Lh))

L:                      gLr  (gLr)  =>  gLr  (   )  =>  (   )
```

New KB:

```
(gFh) (rTh) (rN) (hM) hLr (gFr) (hLg) (rLg) rFh
((10L11)(10L12) 11L12) (gLh) (   ) rLh
((13L14) 14L13) gLr ((10Lr)(10Lh))
```

The boundary deduction has concluded in its characteristic manner by asserting a `( )` into the KB. By the Dominion rule, this truth symbol erases all other forms in the problem space, leaving a mark of proof. Notice in the second to last step, the selection `(gLr)` was also available for binding. It would have been chosen next, should the current choice have failed. And it too would have terminated the proof process.

The signature characteristic of this boundary proof is its minimality. In contrast to resolution, very little search was conducted because the problem was structured as a global statement rather than as a collection of fragments. Thus the available strategies addressed the entire problem at all times. To reconstruct the logic of the boundary proof, we trace the binding processes of steps which are used to reach the conclusion (only the step which generated `rFh` was unnecessary):

```
((4F5)(5M) 5S4)  ((6S7) 6L7)  ==>   ((4F5)(5M) 5L4)          R2+R3      = R23

(rN) ((8F9)(9N) 9L8)          =>  ((8Fr) rL8)               F3+R4      = R4a

(hM) ((4F5)(5M) 5L4)          =>  ((4Fh) hL4)               F4+R23     = R23a

(rTh) ((1F3)(2T3) 1F2)        =>  ((1Fh) 1Fr)               F2+R1      = R1a

(gFh) ((1Fh) 1Fr)             =>  (gFr)                     F1+R1a     = (gFr)

(gFh) ((4Fh) hL4)             =>  (hLg)                     F1+R23a    = (hLg)

(gFr) ((8Fr) rL8)             =>  (rLg)                     (gFr)+R4a  = (rLg)

hLr ((13L14) 14L13)           =>  ((rLh)) => rLh            Q+R6       = rLh

(hLg) ((13L14) 14L13)         =>  (gLh)                     (hLg)+R5   = (gLh)
(rLg) ((13L14) 14L13)         =>  (gLr)                     (rLg)+R5   = (gLr)

hLr ((10L11)(10L12) 11L12)    =>  ((10Lh)(10Lr))            Q+R5       = R5a
rLh ((10L11)(10L12) 11L12)    =>  ((10Lr)(10Lh))            rLh+R5     = R5b

(gLh) ((10Lh)(10Lr))          =>  ((gLr)) => gLr            (gLh)+R5a  = gLr

gLr  (gLr)                                                  gLr+(gLr)  = QED
```

Addendum

The phraseology and structure of rules in a knowledge-base is extremely critical to the success of an inference engine. Examples:

1. To generate a sequence of numbers, it may be tempting to put in a integer generation rule such as

```
if (_1_ is-an-integer) then  ( (_1_ + 1) is-an-integer)
```

This rule could immediately generate an infinite string of integers, which would, of course, be expressed computationally as an over-flow crash.

2. Similar recursive overflows can occur with quite common rules such as transitivity and commutativity. Both of these rules occur in the example KB above (R5 and R6 for the has-same-last-name relation). Implicit in the actual transforms is an "occurs-check" which stops rules from being called when they generate items which duplicate already existing items.

3. Some rules can be expressed in different ways. The forms of these rules strongly effect both the sequence of fact generation, and the efficiency of the deductive process. For instance, transitivity is commonly expressed as

```
if (1R2 and 2R3) then 1R3
```

In the example KB, R5 expresses transitivity as

```
if (1R2 and 1R3) then 2R3
```

This design choice was made because of the natural semantics of the has-same-last-name relation. The design choice has a strong effect on the sequence of generated facts in each example.

4. Some rules implicitly incorporate other rules, in that the other rules are strictly redundant. If we let the variable 3 be equal to "1" in the above transitivity rule, (and we omit the trivial fact that a person has their own last name), we get the commutative R6.

```
if (1R2 and 1R1) then 2R1    ==>    if 1R2 then 2R1
```

Again, the choice of whether to include rules specifically, or let them be implicit in other rules has a strong and unpredictable effect on the performance of the engine.

5. Rules should take care to exclude unwanted cases, although it is often a difficult choice between simple rules with fast cycling time, or larger rules which take effort to compute. This issue also shows up in programming as choices about function decomposition, and in CPU design as RISC vs CISC architectures. In the example, we elected not to exclude the fact that a person has their own last name, but we could have expressed R1 as:

```
if (1F2 and 1≠2 and 2T3 and 2≠3) then 1F3
```

Alternatively, the deduction might have been able to make use of the same-last-name-as-yourself rule, and we may have wanted to include it as

```
      if ( 1 = 2 ) then 1L2
```

or as

```
      if  1L2  then 1L1
      if  1L2  then 2L2
```

These decisions are quite difficult to make, and depend on the expected types of queries, the structure and frequency of facts in the KB, and the other rules in the KB.

6. Rule ordering plays a critical role in algorithmic transformations. When a decision has to be made about what rule to bind next, it is often the case that a general strategy like set-of-support or simplest-first still results in several equally likely choices. Algorithms tend to take the next rule in sequence, but this may not be best or be most efficient. When having to answer a query like "Which people are a child of a President?" it is imperative that the search engine know something about the size of the domains. It is far better to approach this by looking for Presidents first, then looking for their children, than it is to look at all the children in the country and ask each if their parent is a President. You can play with this issue yourself by querying a web search engine for pages which have some common word, such as "set", and some rarer word such as "recursion". Do your results depend on the order of the query words?

7. Finally, the reason for this addendum is that I wasted many hours (and distributed faulty code to the class) with a poorly designed R1. This design flaw was subtle, since both the original and the final form of the rule were valid. R1 originally said "if person A is your father and you are the sister of person B, then person A is the father of person B". This makes sense, but logically it needs the support of another rule, "if you are the sister of person C, then person C is the brother-or-sister of you" in order to converge with the other rules. That is, there was no way for the inference engine to turn around the idea that you are a sister, making it a sibling relation. This inversion was necessary basically due to the structure of the fact-base, in that the constant "Harry" never found its way to a position where it could be matched. The solution in this case was to change the form of R1:

```
      if (1F2 and 2T3) then 1F3            NO
      if (1F3 and 2T3) then 1F2           YES
```

Note that this artifact is due to the very limited rule-base. A more acceptable and correct solution would be to include the entire spectrum of relationships:

```
      if (1 is-sister-of 2) and (2 is-male) then (2 is-brother-of 1)
      if (3 is-brother-of 4) and (4 is-female) then (4 is-sister-of 3)
      if (5 is-father-of 6) and (6 is-sister-of 7) then (5 is-father-of 7)
      if (8 is-father-of 9) and (9 is-brother-of 10) then (8 is-father-of 10)
```