VI
THE DISCRIMINATION-NET

Thus far, c-lists, i-lists, and processed-c-lists have been stored in simple lists (*conclusions*, *interests*, and *processed-conclusions*). Finding a c-list for a formula involves traversing the list. For large databases, this can be made considerably more efficient by storing them instead in a discrimination-net. A discrimination-net is a tree of nodes (d-nodes). Each d-node has a set of *discrimination-tests* associated with it. To classify an object in terms of the discrimination-net, we begin at the top node and apply the tests sequentially until the object satisfies one of the tests. Each test corresponds to a branch descending from the d-node. When an object satisfies a discrimination test, we proceed down the associated branch to the next d-node. This process is repeated until we come to a d-node at which the object passes none of the discrimination tests. This means that the object cannot be further discriminated in terms of the net, so it is classified as being of the type of that final d-node. Typically, objects are stored at the nodes classifying them. Then when we want to find an object of a certain type (characterized by a list of discrimination tests) we can simply walk down the tree to the appropriate d-node rather than searching through a list of all the objects stored.

1. Encoding Formulas

To build a discrimination-net storing formulas (or structures like c-lists and i-lists that are categorized in terms of an associated formula), we need a way of encoding the formula. The resulting formula-profile should be a list of "descriptors". A branch of the net will correspond to a list of descriptors, and a formula will be stored at the node corresponding to the last descriptor in the list. D-nodes will be datastructures encoding the following information:

- d-node number
- discrimination-tests
- d-node-c-lists
- d-node-i-lists
- parent-d-node
- d-node-forwards-reasons

top-d-node will be the node at the top of the discrimination-net.

What remains is to construct an encoding of formulas. This is performed by FORMULA-CODE. FORMULA-CODE returns two values—a list of descriptors, and a list of terms. For example:

```
(formula-code '(-> (r x5 x6) (-> (r x6 x7) (r x5 x7))))
```

returns the values

```
(((1) . ->) ((2 1) . r) ((3 1) . ->) ((3 2 1) . r) ((3 3 1) . r))
(x5 x7 x6 x7 x5 x6)
```

The first value is a list of descriptors, which describe the symbols that occur at specific positions in the formula. For instance, the presence of the descriptor "((3 2 1) . r)" signifies that the first member of the second member of the third member is "r". The description thus produced describes the formula down to the argument-places of the atomic formulas, but does not describe the terms. The second value returned lists the terms. They are listed in order of their occurrence within each atomic subformula, but in reverse order of the occurrence of the subformula in the formula. The first main point of this encoding is that formulas will be categorized first in terms of their main connective. The other main point of the encoding is that to check whether two formulas unify, we need merely check that the first values of their formula-codes are the same, and their term-lists (the second-values) unify. Consequently, if this encoding is used to store formulas in a discrimination-net, the formulas will unify iff they are stored at the same node of the net and their term-lists unify. To take advantage of this, when c-lists and i-lists are stored in the discrimination-net, the term-lists produced by the formula-code of the c-list-formulas and i-list-formulas will be stored in a slot of the c-list or i-list. C-lists and i-lists are also given new slots to record the d-node at which they are stored.

A further feature of FORMULA-CODE is that bound variables are rewritten in a standard form, so that notational variants receive the same code. For example,

```
(formula-code '(all x (some y (r x y))))
```

and

```
(formula-code '(all w (some z (r w z))))
```

produce the same code:

```
(((1) . all) ((2 1) . some) ((2 2 1) . r))
((q-var 1) (q-var 2))
```

2. Storing Inference-Nodes and Interests

Storing an inference-node consists of first searching for a d-node encoding the node-formula. If such a d-node is found, then a search occurs for a c-list stored at that node having that node-formula as the c-list-formula. If no c-list is found, one is created and the node is stored in it. If no d-node is found, then d-nodes are added to the net to create a d-node at which the node can be stored. Storing an interest works similarly. To illustrate, here is an example of a discrimination-net (produced in solving problem 54) generated by this code:

```
|--((1) . V) : #<d-node: 55>
|         |--((2 1) . ~) : #<d-node: 56>
|                   |--((2 2 1) . R) : #<d-node: 57>
|                             |--((3 1) . ~) : #<d-node: 58>
|                                       |--((3 2 1) . R) : #<d-node: 59>
|                                       |      #<c-list for (~(R ^x0 x6) v ~(R x6 ^x0))>
|
|         |--((2 1) . &) : #<d-node: 38>
|         |         |--((2 2 1) . R) : #<d-node: 39>
|         |                   |--((2 3 1) . R) : #<d-node: 40>
|         |                             |--((3 1) . R) : #<d-node: 41>
|         |                             |      #<c-list for (((R x5 x6) & (R x6 x7)) -> (R x5 x7))>
|         |
|--((1) . ->) : #<d-node: 24>
|         |
|         |         |--((3 1) . ~) : #<d-node: 60>
|         |         |         |--((3 2 1) . R) : #<d-node: 61>
|         |         |         |      #<c-list for ((R ^x0 x6) -> ~(R x6 ^x0))>
|         |         |
|         |         |--((3 1) . ->) : #<d-node: 42>
|         |         |         |--((3 2 1) . R) : #<d-node: 43>
|         |         |                   |--((3 3 1) . R) : #<d-node: 44>
|         |         |                   |      #<c-list for ((R x5 x6) -> ((R x6 x7) -> (R x5 x7)))>
|         |         |
|         |--((2 1) . R) : #<d-node: 25>
|                   |--((3 1) . R) : #<d-node: 26>
```

```
        |               |    #<c-list for ((R x6 x7) -> (R (@y2 x6) x7))>
        |               |    #<c-list for ((R (@y2 x5) x7) -> (R x5 x7))>
        |               |    #<c-list for ((R x3 x4) -> (R x4 x3))>
        |
        |--((1) . SOME) : #<d-node: 19>
        |       |--((2 1) . R) : #<d-node: 20>
        |       |    #<c-list for (some y)(R x1 y)>
        |
--#<d-node: 1>
        |
        |       |--((2 1) . &) : #<d-node: 52>
        |       |       |--((2 2 1) . R) : #<d-node: 53>
        |       |               |--((2 3 1) . R) : #<d-node: 54>
        |       |               |    #<c-list for ~((R ^x0 x6) & (R x6 ^x0))>
        |       |
        |       |       |--((2 2 1) . &) : #<d-node: 45>
        |       |       |       |--((2 2 2 1) . R) : #<d-node: 46>
        |       |       |               |--((2 2 3 1) . R) : #<d-node: 47>
        |       |       |                       |--((2 3 1) . R) : #<d-node: 48>
        |       |       |                       |    #<i-list for ~(((R x5 x6) & (R x6 x7)) -> (R x5 x7))>
        |       |       |
        |       |--((2 1) . ->) : #<d-node: 27>
        |       |       |
        |       |       |       |--((2 3 1) . ~) : #<d-node: 62>
        |       |       |       |       |--((2 3 2 1) . R) : #<d-node: 63>
        |       |       |       |       |    #<i-list for ~((R ^x0 x6) -> ~(R x6 ^x0))>
        |       |       |       |
        |       |       |       |--((2 3 1) . ->) : #<d-node: 49>
        |       |       |       |       |--((2 3 2 1) . R) : #<d-node: 50>
        |       |       |       |               |--((2 3 3 1) . R) : #<d-node: 51>
        |       |       |       |               |    #<i-list for ~((R x5 x6) -> ((R x6 x7) -> (R x5 x7)))>
        |       |       |       |
        |       |       |--((2 2 1) . R) : #<d-node: 28>
        |       |               |--((2 3 1) . R) : #<d-node: 29>
        |       |               |    #<i-list for ~((R x3 x4) -> (R x4 x3))>
        |       |
        |--((1) . ~) : #<d-node: 17>
        |       |--((2 1) . R) : #<d-node: 18>
        |       |    #<c-list for ~(R ^x0 (@y2 ^x0))>
        |       |    #<c-list for ~(R (@y2 ^x0) ^x0)>
        |       |    #<c-list for ~(R ^x0 ^x0)>
        |       |    #<i-list for ~(R x1 (@y2 x1))>
        |
        |--((1) . R) : #<d-node: 16>
        |    #<c-list for (R ^x0 ^x0)>
        |    #<c-list for (R (@y2 x3) x3)>
        |    #<c-list for (R x1 (@y2 x1))>
        |    #<i-list for (R ^x0 ^x0)>
        |
        |               |--((2 2 1) . &) : #<d-node: 34>
        |               |       |--((2 2 2 1) . R) : #<d-node: 35>
        |               |               |--((2 2 3 1) . R) : #<d-node: 36>
        |               |                       |--((2 3 1) . R) : #<d-node: 37>
        |               |                       |    #<c-list for (all z)(((R x5 x6) & (R x6 z)) -> (R x5 z))>
        |               |
        |       |--((2 1) . ->) : #<d-node: 21>
        |       |       |--((2 2 1) . R) : #<d-node: 22>
        |       |               |--((2 3 1) . R) : #<d-node: 23>
        |       |               |    #<c-list for (all y)((R x3 y) -> (R y x3))>
        |       |
        |       |--((2 1) . R) : #<d-node: 15>
        |       |    #<i-list for (all x)(R x x)>
        |--((1) . ALL) : #<d-node: 2>
                |
                |       |--((2 2 1) . ALL) : #<d-node: 9>
                |       |       |--((2 2 2 1) . ->) : #<d-node: 10>
                |       |               |--((2 2 2 2 1) . &) : #<d-node: 11>
                |       |                       |--((2 2 2 2 2 1) . R) : #<d-node: 12>
                |       |                           |--((2 2 2 2 3 1) . R) : #<d-node: 13>
                |       |                              |--((2 2 2 3 1) . R) : #<d-node: 14>
                |       |                              | #<c-list for (all x)(all y)(all z)(((R x y) & (R y z)) ->
                |       |                                                                                 (R x z))>
                |       |
                |--((2 1) . ALL) : #<d-node: 5>
                |       |
                |       |       |--((2 2 2 1) . &) : #<d-node: 30>
                |       |       |       |--((2 2 2 2 1) . R) : #<d-node: 31>
                |       |       |               |--((2 2 2 3 1) . R) : #<d-node: 32>
                |       |       |                      |--((2 2 3 1) . R) : #<d-node: 33>
                |       |       |                      |   #<c-list for (all y)(all z)(((R x5 y) & (R y z)) ->
                |         |         |                                              (R x5 z))>
                |         |         |
                |         |--((2 2 1) . ->) : #<d-node: 6>
                |                   |--((2 2 2 1) . R) : #<d-node: 7>
                |                             |--((2 2 3 1) . R) : #<d-node: 8>
                |                             |      #<c-list for (all x)(all y)((R x y) -> (R y x))>
                |
                |--((2 1) . SOME) : #<d-node: 3>
                          |--((2 2 1) . R) : #<d-node: 4>
                          |      #<c-list for (all x)(some y)(R x y)>
```

The discrimination-net is displayed by executing DISPLAY-DISCRIMINATION-NET. Just that part of the net containing d-node number n can be displayed by executing (SHOW-D-NODE n). (SHOW-NODE n) displays that part of the discrimination-net directly relevant to inference-node number n, and (SHOW-INTEREST n) displays that part of the discrimination-net directly relevant to interest number n.

When storing inference-nodes and interests, searches through the existing c-lists and i-lists are made under several circumstances. First, when a c-list is constructed, a search is made for matching i-lists, and when an i-list is constructed, a search is made for matching-c-lists (i.e., i-lists and c-lists whose formulas unify). Previously, this had to be done by searching *conclusions* and *interests*. Now it can be done by just looking at the c-lists and i-lists that are stored at the same d-node, and checking to see whether their term-lists unify. This is the way MATCHING-I-LISTS-FOR and MATCHING-C-LISTS-FOR work. Interest-discharge is then performed in terms of matching-c-lists and matching-i-lists. Second, when an c-list is constructed, a search is made for c-list-contradictors. This is done by finding matching-c-lists for the negation of the c-list-formula. That in turn is done by simply walking down the net to the right d-node, rather than searching through a list of all the c-lists. MAKE-NEW-REDUCTIO-SUPPOSITION, VALIDATING-DEDUCTIVE-NODE, and EI call C-LIST-FOR, which searches for the c-list having a given formula as its c-list-formula. That can now be done by simply walking down the net.

Thus far, the changes described concern c-lists, i-lists, and interest-discharge. They do not concern forwards-reasoning or the discharge of interest-schemes, which proceed in terms of processed-c-lists. On easy problems (problems producing small databases), these changes have no significant effect, neither speeding OSCAR up nor slowing it down. But on hard problems, they have a dramatic effect. For instance, on the Schubert steamroller problem, which involves an inference graph of 411 nodes, these changes reduce the time required to find the proof by 38%.

3. Storing Processed C-lists

When an inference-node is retrieved from the inference-queue for subsequent use in forwards-reasoning and the construction of interest-schemes from mixed-backwards-reasons, it is deemed “processed”. Because only processed inference-nodes are used for this reason, a separate database was previously kept for these nodes. They were stored in “processed-c-lists”, which were in turn stored in the list *processed-conclusions*. This was to keep the search smaller than if it were performed on the list of all *conclusions*. However, this in turn required processed-c-lists and c-lists to be distinct structures. Because the search for c-lists is now so much more efficient, we can cease distinguishing between c-lists and processed-c-lists, and within a single c-list keep two node-lists—one for all c-list-nodes, and one for the c-list-nodes that have been processed.

By storing processed inference-nodes in the discrimination-net, we can make some inference-rules more efficient. For instance, *modus-ponens1* has been written to require a search through *processed-conclusions*. Given a new conditional, we search *processed-conclusions* for a node whose formula unifies with the antecedent. This can now be rewritten to walk down the discrimination-net to the d-node-for the antecedent, check the c-lists stored there to determine which contain processed-nodes, and check those c-lists to determine whether their formulas unify with the antecedent of the conditional. Furthermore, the unification can be made more efficient by unifying the term-lists rather than the whole formulas. *modus-tollens1* and *conditional-modus-tollens* can be made more efficient in the same way.

modus-ponens2 also involves a search of *processed-conclusions*. Given an arbitrary node, *processed-conclusions* is searched for conditionals whose antecedents unifyf with the node-formula. However, this search is harder to eliminate. It can be made somewhat more efficient by just searching under the d-node labeled "(1 . ->)". *modus-tollens2* can be changed similarly. WIth these rules, the Schubert steamroller problem is now done in 53% of the time it took originally.

However, this change to *modus-ponens2* still has us searching through all conditionals rather than just those whose antecedents have the same form as the node-formula of c. To accomplish the latter, we might try constructing the conditional (P -> X), where P is the node-formula of c and X is a variable. Then if we could find a d-node for (P -> X), we could try just searching for c-lists stored at that d-node. This does not quite work. The formula-code for (P -> X) is (((1) . ->) ((2) . p) ((3) . x)). C-lists stored at the d-node for this formlula would have to be conditionals whose consequents were "X". That is not our intent. We want them to be conditionals with arbitrary consequents. That was the point of making X a variable. We can rectify this by encoding formulas with variables somewhat differently. PREMISE-CODE differs from FORMULA-CODE in that when variables are encountered, no descriptor is added to the list of descriptors. Thus (PREMISE-CODE '(-> P X) '(X)) returns (((1) . ->) ((2 1) . p)). The conditionals whose antecedents might unify with P are then those stored at nodes lying *at or below* the d-node for (((1) . ->) ((2 1) . p)) in the discrimination-net. Note that the unification must be done in terms of the formulas rather than the term-lists because the c-list-term-list will contain terms occurring in the consequent of the conditional stored at the c-list, but the term-list produced by PREMISE-CODE will contain no corresponding terms.

There remains a difficulty. Although this works for *modus-ponens2*, the same strategy will not work for *modus-tollens2*. The difficulty is that the profile returned by (PREMISE-CODE '(-> x p) '(x)) is (((1) . ->) ((3) . p)). If we try to walk this description through the net, we will not find a d-node for it. This is because d-nodes correspond to complete formulas. Every complete conditional has an antecedent, and so will have an entry of the form ((2) . ...) preceding ((3) . p). (PURSUE-D-NODE-FOR profile *top-d-node*) takes the descriptors one at a time, and so after finding a node labeled ((1) . ->), it will look for a daughter node labeled ((3) . p) and will not find one. All the daughter nodes will have labels of the form ((2) . ...).

This difficulty arises from the fact that '(-> x p)' is a partially instantiated premise. That is, the second premise of *modus-tollens2* is '(-> x y)' where x and y are premise-variables. Given ~p, the reasoner looks for inference-nodes of the form (-> x p). To find such nodes, it must examine all conditionals, regardless of antecedent, and see whether any of them have p as a consequent. To perform this search within the discrimination-net, when there is a gap in the profile of the partially instantiated premise (e.g., no descriptor beginning with '(2)'), it must move down *all* daughter nodes until it comes to nodes whose descriptors begin with a position-code that is represented in the premise-code. If the descriptor with that position-code disagrees with the corresponding descriptor in the premise-code, further search of that branch of the net can be terminated. If the descriptors agree, the search can continue down that branch until either a disagreement is encountered or the descriptions are exhausted. In the latter case, a d-node has been found whose c-lists are candidates for unification with the partially instantiated premise.

This change reduces the time spent on the Schubert Steamroller Problem by an additional 7%, for an overall reduction to 48% of the original time.

4. Arbitrary Forwards-Reasons

The next task is to construct machinery to treat arbitrary forwards-reasons in the same way *modus-ponens2* and *modus-tollens* are treated above. This involves two procedures. First, when a node is newly retrieved from the inference-queue, we must find all forwards-reasons having a premise that is instantiated by the node-formula. An efficient way to do this is to store the premises themselves in the discrimination-net at the most specific d-node for the premise-code for the premise. For this purpose we add a slot to reasons for *reason-d-nodes*. For example:

```
? (setf simp1
        (make-forwards-reason
         :reason-name "simp"
         :reason-conclusions '(P Q)
         :forwards-premises (list (list '(& P Q) #'is-inference))
         :reason-variables '(P Q)
         ))

? (setf *forwards-reasons* (list simp))

? (initialize-discrimination-net)

? (display-discrimination-net)

--#<d-node: 1>
        |--((1) . &) : #<d-node: 2>
        |      premise: (& p q) for simp
```

Then when a new node is retrieved from the inference-queue, we go to the d-node at which its c-list is stored, and search *upwards* for d-nodes having reason-premises stored at them.

For reasons with reason-functions, like *modus-ponens1*, the search for nodes instantiating premises after the first premise is carried out by the reason-function, so only the first premise should be listed as a forwards-premise for the reason.

Storing the reasons in the discrimination-net will result in a large acceleration of the reasoning when the reasoner is given a large database of reasons. But even on the purely deductive problems using the small array of reasons contained in the list *forwards-logical-reasons*, the harder problems are accelerated by around 5%. This brings the Schubert Steamroller Problem down to 45% of its original time.

When reasoning forwards from an inference-node, the node is used to instantiate the first premise. If there are no more premises, that instantiation will be used to compute the conclusion to be drawn. If there are more premises, the instantiation is used to partially instantiate the next premise. The premise-code for that partially-instantiated premise is computed, and the discrimination-net is searched for nodes instantiating it. This is repeated until all nodes are found instantiating all the premises, and then the cumulative instantiation is used to compute the conclusion to be drawn.

5. A Better Strategy for Forwards-Reasoning—Instantiated Premises

The previous strategy has us do two searches. When a conditional is retrieved from the inference-queue, we search for processed-conclusions that unify with the antecedent; and when any formula is retrieved from the inference-queue, we search for processed conditionals whose antecedents unify with the new formula. The latter search is performed by searching all conditionals. A more efficient strategy is the following. (1) When a conditional is retrieved from the inference-queue, we find the d-node for its antecedent, and search for processed-conclusions the unify with it. This is as before. Note that such conclusions must be stored at the very same d-node. (2) We store the antecedent at that d-node. We can do this by adding a slot *d-node-instantiated-premises* to d-nodes. (3) When any formula is retrieved from the inference-queue, rather than search for conditionals, we just search for conditional-antecedents stored at the d-node-instantiated-premises of that same d-node for unifying antecedents. This latter search should be more efficient than searching through all the conditionals stored in the discrimination-net.

An instantiated-premise will be a data-structure encoding the following information:

- ip-reason
- ip-binding—the binding of the reason-variables produced thus far
- ip-basis—the list of inference-nodes that have instantiated the previous premises
- ip-premise—the first remaining premise

For reasons that will become evident later, we use the same structures for storing forwards-reasons and instantiated-premises.

To illustrate this scheme, when a conditional is retrieved from the inference-queue, APPLY-FORWARDS-REASONS searches upwards from its d-node looking for d-nodes storing applicable forwards-reasons. When the search reaches *conditional-node*, *modus-ponens1* is encountered (stored in d-node-forwards-reasons), and *modus-ponens1* is applied to the conditional.

As *modus-ponens2* imposes no syntactic constraints on formulas to which it applies, it will be stored in the discrimination-net at *top-d-node*. As such, it will not be executed until APPLY-FORWARDS-REASONS searches the discrimination-net all the way up to *top-d-node*. It *could* be applied immediately rather than after performing this search, but that would not actually increase efficiency. This is because the search must take place anyway in the process of looking for other applicable inference-rules. The only effect this will have is to postpone the application of *modus-ponens2* until after other forwards-inference-rules are applied. Nevertheless, to facillitate future generalizations, let us revise the code to apply *modus-ponens2* immediately. The key to doing this is to note that *modus-ponens2* is not really a separate inference-rule—it is just the last half of *modus-ponens2*. So let us delete it from the list of forward-reasons. This will have the effect of no longer storing it at top-d-node*. Let us call *modus-ponens2* a *continutation-function* rather than a reason-function.

To get *modus-ponens2* applied, we revise REASON-FROM-PREMISE-NODE so that it looks not just at d-node-forwards-reasons but also at d-node-instantiated-premises. We can treat *modus-tollens* analogously. These changes result in an acceleration of up to 7% on the test problems (6% on the Schubert Steamroller Problem, and at least 5% on a large number of the other problems).

Next, observe that we need not distinguish between d-node-forwards-reasons and d-node-instantiated-premises. We have used the same structures for recording both, and we can store them both in the same slot in d-nodes.

Storing a forwards-reason at a d-node really amounts to storing its first-premise. We can regard this as the limiting case of storing an instantiated-premise (with an empty instantiation and empty basis). This suggests regarding *modus-ponens1* and *modus-tollens1* as reason-continuations as well. We can treat other forwards-reasons analogously. This has the effect of eliminating all appeal to reason-functions. Alternatively, we can scrap reason-continuations and simply give reason-functions the extra argument "ip".

The significance of this last exercise is that it points the way to a general strategy for dealing with arbitrary forwards-reasons. The first step in the generalization is to add some slots to instantiated-premises:

- ip-condition
- ip-remaining-premises
- ip-instantiations—the list of instantiations of node-variables in the ip-basis.
- ip-used-premise-variables—the premise variables bound thus far.
- derived-premises—instantiated-premises produced by instantiating the present instantiated-premise.
- ip-d-node—the d-node at which the instantiated-premise is stored.

Just as for *modus-ponens2*, COMPUTE-FORWARDS-REASON-D-NODES will now store only the first forwards-premise of a reason in d-node-forwards-reasons. This means that *modus-ponens1* and *modus-tollens1* must be written so that the conditional is the first premise (in fact, they are already written that way). We need not list the other premises, as no use will be made of them (the reason-functions access them implicitly).

Forwards-reasoning will now proceed as follows. First, each forwards-reason will be indexed in the discrimination-net according to its first premise. This will be accomplished by COMPUTE-FORWARDS-REASON-D-NODES. When a new conclusion is processed, we search at and above its d-node for forwards-reasons whose first premises are instantiated by the conclusion (also for instantiated-premises—see below). This search is performed by REASON-FROM-DOMINANT-PREMISE-NODES, which applies REASON-FROM-FIRST-INSTANTIATED-PREMISE to the d-node of the conclusion and all d-nodes above it in the discrimination-net. REASON-FROM-FIRST-INSTANTIATED-PREMISE looks at each instantiated-premise of a d-node. If the ip-reason has a reason-function, that is applied as in *modus-ponens1* and *modus-ponens2*. If the ip-reason has no reason-function and the instantiated-premise has an empty ip-basis, REASON-SUBSTANTIVELY-FROM-FIRST-INSTANTIATED-PREMISE is applied. (If the ip-basis is nonempty, REASON-SUBSTANTIVELY-FROM-NON-INITIAL-INSTANTIATED-PREMISE is applied, as described below.

REASON-SUBSTANTIVELY-FROM-FIRST-INSTANTIATED-PREMISE first checks to see whether the processed conclusion instantiates the instantiated-premise. This check is carried out by first pattern-matching the node-formula to the premise-formula, using the premise-variables as the pattern variables. If a premise-variable occurs more than once in the premise-formula, it may correspond to more than one term in the node-formula. This correspondence is recorded in a *term-list*, which is an a-list of dotted pairs (var . terms) where var is a premise-variable and terms is the list of terms in the node-formula corresponding to occurrences of var in the premise-formula. For the node to instantiate the premise, what is required is that there be a unifier *u* that unifies all members of terms for each (var . terms) in the term-list. *u* will be the instantiation for the node-formula. Applying u to the term-list produces a binding of the premise-variables, consisting of an a-list of dotted-pairs (var . term) where term is the result of unifying all members of terms (in (var . terms)) using *u*. With each premise we will store a two-valued function (called a binding-function) that produces the binding and the instantiation when applied to an arbitrary node-formula and node-variables. Thus the instantiation check is carried out by applying the premise binding-function to the node-formula and node-variables. If the instantiation *u* is non-nil, the node instantiates the premise.

If the node instantiates the premise and there are no more premises, then an inference is made in accordance with the (monadic) forwards-reason. If there are more premises, then the binding is applied to the second premise (producing a partially instantiated premise) and an instantiated-premise is constructed for it and stored in the discrimination-net. The ip-binding is the binding and the list of ip-instantiations is the list whose only member is *u*. The ip-basis is the unit set of the node. A search is carried out over the d-nodes at and below which the instantiated-premise is stored for processed-c-lists whose c-list-nodes instantiate the new instantiated-premise. The search proceeds by applying REASON-SUBSTANTIVELY-FROM-NON-INITIAL-INSTANTIATED-PREMISE to the instantiated-premise and each processed-c-list.

REASON-SUBSTANTIVELY-FROM-NON-INITIAL-INSTANTIATED-PREMISE proceeds by first checking to see whether the c-list-nodes instantiate the premise. It does this by first applying the premise binding-function to the c-list-formula to produce a binding (binding0) and an instantiation (instantiation0). If instantiation0 is non-nil, then it checks to see whether binding0 and instantiation0 are consistent with the ip-binding and ip-instantiation. It does this by looking for a unifier *u* that simultaneously unifies the terms the two bindings assign to common premise-variables. Given a non-nil *u*, it is applied to binding0 and the ip-binding to produce a new binding *binding*. Where *instantions0* is the list of instantiations for the prior premises, let *instantiations1* be the result of sequentially merging *u2* with these instantiations, and let *instantiation* be the instantiation that results from sequentially merging *u1* with instantiation0. Let *instantiations* be the result of consing *instantiation* to the front of *instantiations1*. If there are no more premises, then an inference is made in accordance with the forwards-reason. If there are more premises, then *binding* is applied to the next premise and an instantiated-premise is constructed for it and stored in the discrimination-net. The ip-binding is *binding* and the ip-instantiations is *instantiation*s. The ip-basis is the result of adding the processed-c-list to the front of the previous ip-basis. A search is carried out over the d-nodes at and below which the instantiated-premise is stored for processed-c-lists whose c-list-nodes instantiate the new instantiated-premise.

The search proceeds by applying REASON-SUBSTANTIVELY-FROM-NON-INITIAL-INSTANTIATED-PREMISE to the instantiated-premise and each processed-c-list.

As noted above, when a new processed conclusion is retrieved from the inference-queue, REASON-FROM-INSTANTIATED-PREMISES also applies REASON-SUBSTANTIVELY-FROM-NON-INITIAL-INSTANTIATED-PREMISE when an instantiated-premise with a non-empty ip-basis is found at or above the new node in the discrimination-net. In that case, if REASON-SUBSTANTIVELY-FROM-NON-INITIAL-INSTANTIATED-PREMISE produces a new instantiated-premise, the new processed-conclusion is inserted into the ip-basis rather than the processed-c-list for for the conclusion. In this way, ip-bases come to be lists containing both individual inference-nodes and processed-c-lists. When an inference is made (upon instantiating all the reason-premises), it is made from every basis (list of inference-nodes) that can be constructed from the ip-basis.

All parts of the discrimination-net containing nodes directly relevant to a forwards-reason can be displayed by executing (SHOW-REASON *reason*).

6. Backwards-Reasoning

The discrimination-net can be used to make backwards-reasoning more efficient in much the same way it is used for forwards-reasoning. First, backwards-reasons can be stored in the discrimination-net according to their conclusions, and then when an interest is adopted, applicable backwards-reasons can be found by searching at or above the interest in the discrimination-net.

For simple-backwards-reasons (those wth no forwards-premises), REASON-SUBSTANTIVELY-FROM-BACKWARDS-REASON simply finds the binding of the premise-variables provided by instantiating the reason-consequent by the interest-formula, and then builds an interest-link to interest in the backwards-premises as instantiated by the binding. In simple cases, the binding results from pattern-matching the interest-formula and the reason-consequent, but if a premise-variable has multiple occurrences in the reason-conclusion, then constructing the binding requires unifying the terms corresponding to the different occurrences. This process is the same as that encountered in forwards-reasoning. To make this efficient, it will be convenient to store a binding-function and a list of the premise-variables occurring in the reason-consequent in the backwards-reason:

Simple backwards-reasons have no forwards-premises, and as such require no special treatment in connection with the discrimination-net. However, mixed and degenerate backwards-reasons have forwards-premises that are instantiated by processed-conclusions. When a new interest of the appropriate form is adopted, it is used to partially instantiate the premises of the reason, creating a new interest-scheme that is added to *interest-schemes*. The previous code then required a search of *processed-conclusions* to find any that instantiate the forwards-premises of the interest-scheme. In addition, whenever a conclusion is retrieved from the inference-queue, a search was made of *interest-schemes* to find any the conclusion instantiates. Both of these searches can be done more efficiently using the discrimination-net. This is accomplished by storing the partially instantiated forwards-premises at d-nodes in essentially the same way the forwards-premises for forwards-reasons are stored. When a new interest-scheme is adopted, and its forwards-premises are stored in the discrimination-net, the search for conclusions instantiating the premises can be restricted to d-nodes *at or below* the d-nodes at which the premises are stored, and when a new conclusion is retrieved from the inference-queue, the search for interest-schemes it might instantiate can proceed by searching *at or above* it in the discrimination-net for those forwards-premises of interest-schemes. Discharging the interest-scheme works like REASON-FROM-PREMISE-NODE.

To implement this, we add a slot to d-nodes for *d-node-interest-schemes*. CONSTRUCT-INTEREST-SCHEME will work like STORE-FORWARDS-REASON, instantiating the forwards-reasons by applying the binding already obtained by matching the reason-consequent with the interest-formula. Interest-schemes will be just like instantiated-premises, except that they have an additional slot for *target-interest* and for *instance-function*, where the latter is a function that, when applied to an interest-formula of the appropriate logical form, produces the instantiation of the premise-variables that produces the interest-formula from the (first) reason-conclusion.

- is-reason
- is-binding—the binding of the reason-variables produced thus far.
- is-basis—the list of inference-nodes that have instantiated the previous premises.
- is-premise—the first remaining premise.
- is-condition
- is-remaining-premises
- is-instantiations—the list of instantiations of node-variables in the is-basis.
- is-used-premise-variables—the premise variables bound thus far.
- derived-premises—instantiated-premises produced by instantiating the present instantiated-premise.
- is-d-node—the d-node at which the instantiated-premise is stored.
- is-target-interest
- is-instance-function

When an interest-scheme is stored at a d-node, discharge-interest-scheme will search at or below the d-node for processed-conclusions that instantiate the first premise. REASON-FROM-INTEREST-SCHEME will be analogous to REASON-SUBSTANTIVELY-FROM-NON-INITIAL-PREMISE. DISCHARGE-INTEREST-SCHEMES is called when an inference-node is retrieved from the inference-queue. This can now be made analogous to APPLY-FORWARDS-REASONS.

7. Cancellation

When inference-nodes and interests are cancelled, this can result in the emptying of i-list-interests for i-lists and c-list-nodes for c-lists. When this happens, they should be removed from their d-nodes. If this leaves the d-nodes without c-lists or i-lists, and they have no discrimination-tests, d-node-forwards-reasons, d-node-backwards-reasons, or d-node-interest-schemes, then they become pointless and in the interest of efficiency they should be removed from the discrimination-net.

When an interest is cancelled, this should cancel interest-schemes for which it is the target-interest. To implement this, a record must be kept in interests of the generated-interest-schemes. Then when an interest is cancelled, all generated-interest-schemses will be cancelled, together with all interest-schemes derived from them.

Similarly, when an inference-node is cancelled, if this leaves its c-list without any processed-nodes, this should cancel instantiated-premises and interest-schemes in whose basis the node or c-list occurs. To implement this, c-lists must have slots for *generated-instantiated-premises* and *supported-interest-schemes*. Then when a c-list is left without any processed-nodes, all supported-interest-schemes will be cancelled, together with all derived-interest-schemes, and all generated-instantiated-premises will be cancelled, together with all derived-premises.

8. An Example

We have added a number of slots to d-nodes. The full structure of the discrimination-net can now be illustrated by returning to the example given earlier of problem 54:

```
|--((1) . R) : #<d-node: 37>
|       #<c-list for (R x5 x5)>
|       #<c-list for (R (@y2 x6) (@y2 x6))>
|       #<c-list for (R (@y2 (@y2 x7)) x7)>
|       #<c-list for (R (@y2 x3) x3)>
|       #<c-list for (R x1 (@y2 x1))>
|       instantiated-premise 31 for modus-ponens2: (R (#:|@y2| #:\x5) #:\x7)
|       instantiated-premise 29 for modus-ponens2: (R #:\x6 #:\x7)
|       instantiated-premise 27 for modus-ponens2: (R #:\x5 #:\x6)
|       instantiated-premise 23 for modus-ponens2: (R #:\x3 #:\x4)
```

```
        |--((1) . some) : #<d-node: 21>
        |    first premise for EI: (some X)P
        |    conclusion for EG
        |       |--((2 1) . R) : #<d-node: 38>
        |       |    #<c-list for (some y)(R x1 y)>
        |
        |               |--((2 2 1) . &) : #<d-node: 51>
        |               |       |--((2 2 2 1) . R) : #<d-node: 52>
        |               |               |--((2 2 3 1) . R) : #<d-node: 53>
        |               |                       |--((2 3 1) . R) : #<d-node: 54>
        |               |                       |    #<c-list for (all z)(((R x5 x6) & (R x6 z)) -> (R x5 z))>
        |               |
        |       |--((2 1) . ->) : #<d-node: 40>
        |       |       |--((2 2 1) . R) : #<d-node: 41>
        |       |               |--((2 3 1) . R) : #<d-node: 42>
        |       |               |    #<c-list for (all y)((R x3 y) -> (R y x3))>
        |       |
        |       |--((2 1) . R) : #<d-node: 36>
        |       |    #<c-list for (all x)(R x x)>
        |--((1) . all) : #<d-node: 20>
        |    first premise for UI: (all X)P
        |    conclusion for UG
        |       |
        |       |       |--((2 2 1) . all) : #<d-node: 30>
        |       |       |       |--((2 2 2 1) . ->) : #<d-node: 31>
        |       |       |               |--((2 2 2 2 1) . &) : #<d-node: 32>
        |       |       |                       |--((2 2 2 2 2 1) . R) : #<d-node: 33>
        |       |       |                               |--((2 2 2 2 3 1) . R) : #<d-node: 34>
        |       |       |                                       |--((2 2 2 3 1) . R) : #<d-node: 35>
        |       |       |                                       |    #<c-list for (all x)(all y)(all z)(((R x y)
        |       |       |                                                & (R y z)) -> (R x z))>
        |       |       |
        |       |--((2 1) . all) : #<d-node: 26>
        |       |       |
        |       |       |       |--((2 2 2 1) . &) : #<d-node: 47>
        |       |       |       |       |--((2 2 2 2 1) . R) : #<d-node: 48>
        |       |       |       |               |--((2 2 2 3 1) . R) : #<d-node: 49>
        |       |       |       |                       |--((2 2 3 1) . R) : #<d-node: 50>
        |       |       |       |                       |    #<c-list for (all y)(all z)(((R x5 y) & (R y z)) ->
        |       |       |       |                                                       (R x5 z))>
        |       |       |       |
        |       |       |--((2 2 1) . ->) : #<d-node: 27>
        |       |               |--((2 2 2 1) . R) : #<d-node: 28>
        |       |                       |--((2 2 3 1) . R) : #<d-node: 29>
        |       |                       |    #<c-list for (all x)(all y)((R x y) -> (R y x))>
        |       |
        |       |--((2 1) . some) : #<d-node: 24>
        |               |--((2 2 1) . R) : #<d-node: 25>
        |               |    #<c-list for (all x)(some y)(R x y)>
        |
        |--((1) . &) : #<d-node: 17>
        |    first premise for simp: (P & Q)
        |    conclusion for adjunction
        |       |--((2 1) . R) : #<d-node: 61>
        |               |--((3 1) . R) : #<d-node: 62>
        |               |    instantiated-premise 25 for modus-ponens2: ((R #:\x5 #:\x6) & (R #:\x6 #:\x7))
        |
--#<d-node: 1>
        |
        |       |--((2 1) . R) : #<d-node: 39>
        |       |    instantiated-premise 30 for modus-tollens2: ~(R (#:|@y2| #:\x6) #:\x7)
        |       |    instantiated-premise 26 for modus-tollens2: ~(R #:\x5 #:\x7)
        |       |    instantiated-premise 24 for modus-tollens2: ~(R #:\x4 #:\x3)
        |       |--((2 1) . all) : #<d-node: 23>
        |       |    first premise for neg-ug: ~(all X)P
        |       |    conclusion for i-neg-ug
        |       |--((2 1) . some) : #<d-node: 22>
        |       |    first premise for neg-eg: ~(some X)P
        |       |    conclusion for i-neg-eg
        |       |--((2 1) . ~) : #<d-node: 16>
        |       |    first premise for neg-elim: P
        |       |    conclusion for neg-intro
        |--((1) . ~) : #<d-node: 11>
        |       |--((2 1) . V) : #<d-node: 15>
        |       |    first premise for neg-disj: ~(P v Q)
        |       |    conclusion for i-neg-disj
        |       |--((2 1) . ->) : #<d-node: 14>
        |       |    first premise for neg-condit: ~(P -> Q)
        |       |    conclusion for i-neg-condit
        |       |       |--((2 2 1) . R) : #<d-node: 45>
        |       |               |--((2 3 1) . R) : #<d-node: 46>
        |       |               |   instantiated-premise 28 for modus-tollens2: ~((R #:\x6 #:\x7) ->
        |       |                                                                   (R #:\x5 #:\x7))
        |       |
        |       |--((2 1) . <->) : #<d-node: 13>
        |       |   first premise for neg-bicondit-simp: ~(P <-> Q)
        |       |   conclusion for i-neg-bicondit
        |       |--((2 1) . &) : #<d-node: 12>
        |       |   first premise for DM: ~(P & Q)
        |       |   conclusion for i-DM
        |
        |--((1) . <->) : #<d-node: 10>
        |   first premise for bicondit-simp: (P <-> Q)
        |   conclusion for bicondit-intro
        |--((1) . V) : #<d-node: 6>
        |   first premise for disj-simp: (P v Q)
        |   conclusion for disj-cond-2
        |   conclusion for disj-cond
        |
        |               |--((3 1) . ->) : #<d-node: 58>
        |               |       |--((3 2 1) . R) : #<d-node: 59>
        |               |               |--((3 3 1) . R) : #<d-node: 60>
        |               |               |   #<c-list for ((R x5 x6) -> ((R x6 x7) -> (R x5 x7)))>
        |               |
        |       |--((2 1) . R) : #<d-node: 43>
        |       |       |--((3 1) . R) : #<d-node: 44>
        |       |       |   #<c-list for ((R x6 x7) -> (R (@y2 x6) x7))>
        |       |       |   #<c-list for ((R (@y2 x5) x7) -> (R x5 x7))>
        |       |       |   #<c-list for ((R x3 x4) -> (R x4 x3))>
        |       |
        |       |--((2 1) . some) : #<d-node: 19>
        |       |   first premise for E-removal: ((some X)P -> Q)
        |       |--((2 1) . all) : #<d-node: 18>
        |       |   first premise for A-removal: ((all X)P -> Q)
        |       |--((2 1) . &) : #<d-node: 9>
        |       |   first premise for exportation: ((P & Q) -> R)
        |       |       |--((2 2 1) . R) : #<d-node: 55>
        |       |               |--((2 3 1) . R) : #<d-node: 56>
        |       |                       |--((3 1) . R) : #<d-node: 57>
        |       |                       |   #<c-list for (((R x5 x6) & (R x6 x7)) -> (R x5 x7))>
        |       |
        |--((1) . ->) : #<d-node: 2>
        |   first premise for modus-ponens1: (%p -> %q)
        |   first premise for modus-tollens1: (%p -> %q)
        |   first premise for conditional-modus-tollens: (P -> Q)
        |   first premise for cond-simp1: (P -> ~P)
        |   conclusion for conditionalization
        |       |--((2 1) . V) : #<d-node: 8>
        |       |   first premise for disj-antecedent-simp: ((P v Q) -> R)
        |       |--((2 1) . ->) : #<d-node: 7>
        |       |   first premise for cond-antecedent-simp: ((P -> Q) -> R)
        |       |--((2 1) . ~) : #<d-node: 5>
        |       |   first premise for cond-simp2: (~P -> P)
        |
        |--((1) . @) : #<d-node: 3>
                |--((2 1) . &) : #<d-node: 4>
```