III
EPISTEMIC COGNITION

1. The Overall Structure

OSCAR receives two kinds of input — percepts provided by perception, and desires produced by optative dispositions. Epistemic cognition produces beliefs about adoptable plans, as discussed above, and these are used to initiate action. Both inputs and the results of inference are stored as nodes of the *inference-graph*. They are also placed on the *inference-queue*, which is a queue of inferences waiting to be performed, as discussed above. It will be discussed further below. For use as a free-standing reasoner, OSCAR can be given unsupported premises in the list *premises*. Premises are triples ⟨*formula,background-knowledge?,strength*⟩, where *formula* is the premise-formula and *strength* is the undefeated-degree-of-support posited for the sequent built out of the supposition and formula. If *strength* is 1, the premise is assumed to be a necessary truth, and is used as such in deductive reasoning. If *strength* is not 1, the premise can be defeated by finding a rebutting defeater, but no sense can be made of undercutting defeat for premises. *Background-knowledge?* is T or NIL, and is used to control reasoning. This will be discussed further in Chapter Four.

This general architecture is encoded as follows:

OSCAR:
- Insert the members of *permanent-ultimate-epistemic-interests* into *ultimate-epistemic-interests*, and insert them into the *inference-queue*.
- Insert the members of *premises* into the *inference-queue*, adopting interest in rebutting defeaters for those that are not necessary truths.
- Initialize *global-assignment-tree*.
- Do the following repeatedly and in parallel:
    - Run environmental input through the optative dispositions, insert the resulting desires into the *inference-queue*, and encode this input in nodes of the *inference-graph*. Update the desire-strengths of desires already on the *inference-queue*, and adjust their priorities in light of changes in desire-strength.
    - Insert new perceptual states into the *inference-queue* and encode this input in nodes of the *inference-graph*. Update the clarities of percepts already on the *inference-queue* and adjust their priorities in light of changes in clarity.
    - THINK
    - INITIATE-ACTIONS

For use as a free-standing reasoner that is not embedded in an agent, COGITATE is a variant of OSCAR that does not deal with percepts, desires, or actions:

COGITATE:
- Insert the members of *permanent-ultimate-epistemic-interests* into *ultimate-epistemic-interests*, and insert them into the *inference-queue*.
- Insert the members of *premises* into the *inference-queue*, adopting interest in rebutting defeaters for those that are not necessary truths.
- Initialize *global-assignment-tree*.
- Do the following repeatedly:
    - THINK-OR-DIE

THINK-OR-DIE differs from THINK only in that when the *inference-queue* becomes empty, the reasoner stops. It is COGITATE rather than OSCAR that is called by the function TEST, whose use is described in chapter one. SIMULATE-OSCAR is a more complex variant of OSCAR that siulates perceptual input. It will be described below.

The *inference-graph* is the data-structure recording the reasoner's forwards-reasoning. In the interest of theoretical clarity, I defined the *inference-graph* in chapter two in such a way that different arguments for the same conclusion are represented by different nodes. This made it clearer how the algorithm for computing defeat-status works. However, for the purpose of implementing defeasible reasoning, this is an inefficient representation of the reasoning, because it leads to needless duplication. If we have two arguments supporting a single conclusion, then any further reasoning from that conclusion will generate two different nodes. If we have two arguments for each of two conclusions, and another inference proceeds from those two conclusions, the latter will have to be represented by four different nodes in the inference-graph, and so on. This is illustrated in figure 1, where *P* and *Q* are each inferred in two separate ways, and then *R* is inferred from *P* and *Q*.

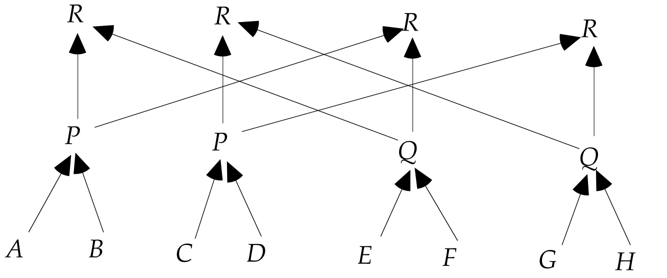

Figure 1. An inference-graph with multiple arguments for a single conclusion.

A more efficient representation of reasoning would take the *inference-graph* to be an and/or graph rather than a standard graph. In an and/or graph, nodes are linked to *sets* of nodes rather than individual nodes. This is represented diagramatically by connecting the links with arcs. In an and/or inference-graph, when we have multiple arguments for a conclusion, the single node representing that conclusion will be tied to different bases by separate groups of links. This is illustrated in figure 2 by an and/or inference-graph encoding the same reasoning as the standard inference-graph in figure 1.

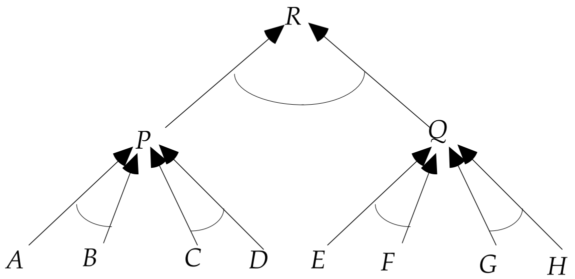

Figure 2. An and/or Inference-graph.

In an and/or inference-graph, a *support-link* will be a *set* of supporting-arrows connected with an arc.

Although and/or graphs provide an efficient representation of reasoning, they complicate the computation of defeat-status. Using simple inference-graphs, we can use the defeat-status computation of chapter two to compute defeat-statuses, and then use the computed defeat-statuses to compute the undefeated-degrees-of-support for the nodes in the inference-graph. The undefeated-degree-of-support for a node is zero if it is defeated, and it is its strength otherwise.

With and/or inference-graphs, we can no longer separate the computation of defeat-statuses and undefeated-degrees-of-support. To illustrate, consider the two inference-graphs in figures 3 and 4. They could arise from different degrees of support for the two nodes supporting *R*. In figure 3, only the rightmost node is strong enough to defeat *S*, whereas in figure 4, only the leftmost node is strong enough to defeat *S*. The difference has repurcusions, because in figure 3, *S* is provisionally defeated, and hence *T* is also provisionally defeated, but in figure 4, *S* is defeated outright, and hence *T* is undefeated. The difficulty is now that if we try to represent these as and/or graphs, we get the same graph, drawn in figure 5. This representation cannot distinguish between the situation diagrammed in figure 3 and that diagrammed in figure 4
.

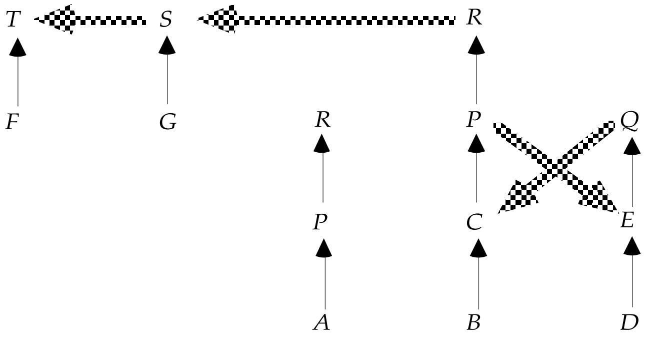

Figure 3. *T* is provisionally defeated.

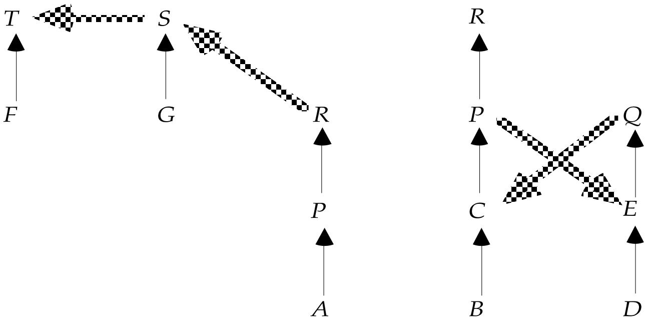

Figure 4. *T* is undefeated.

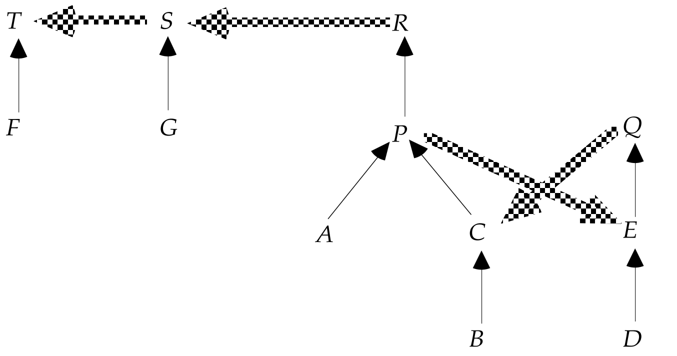

Figure 5. The status of *T* is indeterminate.

This difficulty can be avoided by attaching strengths to the support-arrows. In figure 6, if we attach the strengths in brackets, we get a representation of figure 3, and if we attach the strenghts in braces, we get a representation of figure 4.

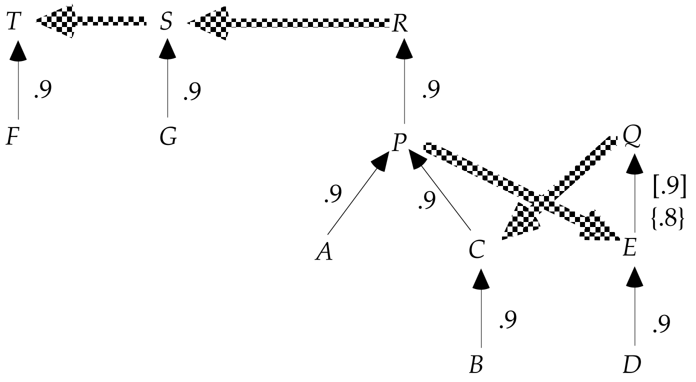

Figure 6. The status of *T* varies.

With this modification, we can construct a defeat-status computation that is equivalent to that for simple inference-graphs. The trick is to take the defeat-status of a node to be its undefeated-degree-of-support rather than just "defeated" or "undefeated". If all arguments supporting a node are defeated, then the undefeated-degree-of-support is 0. Otherwise, it is the maximum of the strengths of the undefeated arguments. We can then modify the definition given in chapter two as follows:

> A node of the *inference-graph* is *initial* iff its node-basis and list of node-defeaters is empty.
>
> σ is a *partial status assignment* iff σ is a function assigning real numbers between 0 and 1 to a subset of the nodes of an inference-graph and to the support-links of those nodes in such a way that:
> 1. σ assigns its node-strength to any initial node;
> 2. If σ assigns a value α to a defeat-node for a support-link and assigns a value less than or equal to α to some member of the link-basis, then σ assigns 0 to the link;
> 3. Otherwise, if σ assigns values to every member of the link-basis of a link and every link-defeater for the link, σ assigns to the link the minimum of the strength of the link-rule and the numbers σ assigns to the members of the link-basis.
> 4. If every support-link of a node is assigned 0, the node is assigned 0;

5. If some support-link of a node is assigned a value greater than 0, the node is assigned the maximum of the values assigned to its support-links.
6. If every support-link of a node that is assigned a value is assigned 0, but some support-link of the node is not assigned a value, then the node is not assigned a value.

> σ is a *status assignment* iff σ is a partial status assignment and σ is not properly contained in any other partial status assignment

Now the question arises how defeat-statuses (or undefeated-degrees-of-support) should be determined for the nodes of the inference-graph. The following proposal seems natural:

> The *undefeated-degree-of-support* of a node of the and/or inference-graph is 0 if some status-assignment fails to assign a value to it; otherwise it is the minimum of the values assigned to it by status assignments.

However, this way of computing undefeated-degrees-of-support for nodes of an and/or inference-graph is not equivalent to the preceding method of computing degrees-of-jusitification for a simple inference-graph. Equivalence would require the truth of the following *Correspondence Theorem*:

> A node of the and/or inference-graph is undefeated (has a nonzero undefeated-degree-of-support) iff one of the corresponding nodes of the simple inference-graph is undefeated.

The above proposal for computing undefeated-degrees-of-support does not make the Correspondence Theorem true. Figure seven is a simple counterexample, with the simple inference-graph on the right and the corresponding and/or graph on the left. In the simple graph, there are two status-assignments, and one assigns "undefeated" to the left "*E*" and the other assigns "undefeated" to the right "*E*", but neither "*E*" is assigned "undefeated" by both status-assignments, so both are defeated. In the and/or graph, there are also two status-assignments, and each assigns "undefeated" to "*E*" (by making a different argument undefeated). Thus on the above proposal, "*E*" would be undefeated in the and/or inference-graph, but defeated in the simple inference-graph.

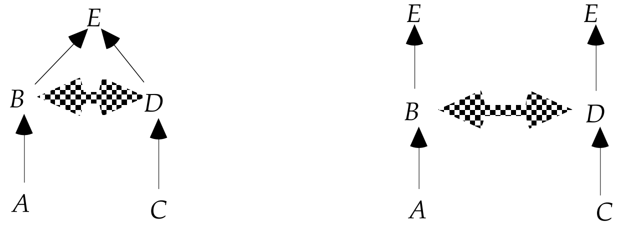

Figure 7. A counterexample to the Correspondence Theorem.

This example makes it clear that the difficulty lies in the fact that, in the and/or graph, there is no argument that comes out undefeated on both assignments. Each simple inference-graph can be rewritten as an and/or graph, and then each node of the and/or graph corresponds to a set of nodes of the simple graph. A node of the simple inference-graph, on the other hand, corresponds to an *argument* in the and/or graph. An argument is a kind of connected sub-tree of the graph. More precisely:

> An *argument from* an and/or inference-graph G *for* a node *N* is a subset A of the nodes and support-links of the graph such that (1) if a node in A has any support-links in G, exactly one of them is in A, (2) if a support-link is in A then the nodes in its support-link-basis are also in A, and (3) *N* is in A.

Nodes in the simple inference-graph correspond one-one to arguments in the and/or inference-graph. The definition of a partial status-assignment for and/or graphs just amounts to saying that a node is assigned "undefeated" if it is either an initial node or some argument for it consists of nodes and links that are assigned "undefeated", and it is assigned "defeated" if it is not an initial node and every argument for it contains a link that is assigned "defeated". This suggests that the correct analysis of defeat-status for and/or inference-graphs should be:

> An argument is *undefeated relative to an and/or inference-graph* iff every status assignment assigns a non-zero value to all nodes and links in the argument.
>
> The *strength* of an argument is the minimum of the node-strengths of the members of *input* contained in it and the reason-strengths of the support-link-rules of the support-links contained in it.
>
> The undefeated-degree-of-support of a node of an and/or inference-graph is the maximum of the strengths of the undefeated arguments for it.

With this definition it becomes simple to prove the Correspondence Theorem by induction on the length of the arguments. To compute undefeated-degrees-of-support efficiently, we can store the list of arguments for a node along with the node, however computing and updating the lists of node-arguments becomes computationally expensive in complex inference-graphs. We can instead do the computation recursively without constructing node-arguments. To see how this works, first consider the simplified case where all strengths are 1 or 0. In that case, we can just talk about nodes being defeated (having undefeated-degree-of-support 0) or undefeated (having undefeated-degree-of-support 1.0). This can be characterized recursively as follows:

> A node is undefeated iff either (a) it is an initial node, or (b) it has a support-link that (1) has no defeating status-assignment and (2) has a basis consisting of undefeated-nodes.

Generalizing this to attach arbitrary values to strengths, undefeated-degrees-of-support can be characterized recursively as follows:

> The undefeated-degree-of-support of an initial node is 1. The undefeated-degree-of-support of a non-initial node is the maximum, over those of its support-links having no defeating status-assignments, of the minimum of the support-link-strength and the undefeated-degrees-of-support of its basis; or 0 if it has no support-links without defeating-assignment-trees.

This allows us to compute undefeated-degrees-of-support recursively as follows.

(1) Find all support-links that have lost or acquired defeating status-assignments. This is the set of altered-links.

(2) Let N_0 be the set of support-link-targets for the altered-links, and let *N* be the set of inference-descendants of N0. For each member of *N*, set its undefeated-degree-of-support in terms of those of its support-links that do not have defeating status-assignments and whose bases are disjoint from *N* (set it to 0 if there are none).

(3) For each node in *N*, if it has a nonzero undefeated-degree-of-support, then for each of its consequent-links *L* that have no defeating status-assignments, compute a undefeated-degree-of-support for the target of *L* as the minimum of the support-link-strength of *L* and the undefeated-degrees-of-support of its basis. If that is greater than the existing undefeated-degree-of-support, recurse that change through the node-descendants of the target using altered-links.

With the understanding that defeat-statuses will be computed as described, the *inference-graph* will consist of a set of inference-nodes and support-links. A support-link is a data-structure encoding an inference, and it has slots for the following information:

- support-link-target — the inference-node supported by the link;
- support-link-basis — the list of inference-nodes from which the inference is made;
- support-link-rule — the reason licensing the inference; this can be either a substantive reason or a keyword naming a reason;
- defeasible? — T if the inference is a defeasible one, and NIL otherwise;
- support-link-defeaters — if the inference is defeasible, the set of inference-nodes having the syntactically appropriate form to defeat it (this ignores considerations of strength);
- defeating-assignment-trees — explained below;
- support-link-discount-factor — used in prioritizing reasoning, and discussed below in connection with the *inference-queue*;
- support-link-nearest-defeasible-ancestors — defined below;
- support-link-strength — a number between 0.0 and 1.0;
- support-link-conclusive-defeat-status — T if there is a deductive node that defeats the link, NIL otherwise.

An inference-node *N* is a data-structure encoding the following information:

- the node-sequent — in the most common case where *N* encodes the conclusion of an inference, this is the sequent inferred; in some cases *N* will encode nondoxastic input, specifically percepts and desires, and their contents will be encoded as sequents;
- the node-formula — the sequent-formula of the node-sequent;
- the node-supposition — the sequent-supposition of the node-sequent;
- the node-kind — :inference, :percept, or :desire;
- support-links — the support-links supporting the node;
- node-justification — if the node has no support-links, this is a keyword describing its justification;
- consequent-links — the list of support-links for which this node is a member of the support-link-basis;
- old-undefeated-degree-of-support — the undefeated-degree-of-support prior to the last computation of defeat-statuses;
- maximal-degree-of-support — the maximal strength of the support-links, used to verify that a node has adequate support for an interest-discharge;
- node-defeatees — the list of support-links *L* such that *N* is a node-defeater of *L*;
- the undefeated-degree-of-support — the undefeated degree-of-support;
- node-ancestors — the list of inference-nodes used in arguing to this node;
- nearest-defeasible-ancestors — the list of node-ancestors inferred by reasoning defeasibly and from which all subsequent reasoning to *N* is deductive;
- answered-queries — the list of queries (see below) answered by the node;
- generated-interests — if the node encodes a supposition, this is the list of interests generated from it;
- generating-interests — if the node encodes a supposition, this is the list of interests that gave rise to the supposition;
- cancelled-node — T if *N* has been cancelled, NIL otherwise;
- discounted-node-strength — used in prioritizing reasoning, and discussed below in connection with the *inference-queue*;
- processed? — T if the node has been processed (see below), NIL otherwise;
- discharged-interests — lists the interests discharged by the conclusion;
- interests-discharged? — T if the conclusion has been used to discharge interests, and NIL otherwise;
- node-c-list — explained below;
- node-queue-node — if the node is currently on the *inference-queue*, this lists the

queue-node enqueing it;

- enabling-interests — if the node is inferred by discharging interest-links, this is the list of resultant-interests of those interest-links;
- generated-defeat-interests — the list of interests in defeaters for the inference to *N*.

Using simple inference-graphs, the list of nearest-defeasible-ancestors of a node *N* is the list of the first defeasible nodes reached by working backwards along the support-links of *N* (or *N* itself if *N* is a defeasible node). In an and/or inference-graph a single node represents a set of nodes from a simple inference-graph, and accordingly, it must have a set of lists of nearest-defeasible-ancestors. The slot *nearest-defeasible-ancestors* holds that set of lists. This set of lists is computed as follows:

For each support-link *L* for *N*:

- if *L* represents a defeasible inference, then {*N*} is is the only member of the list of support-link -nearest-defeasible-ancestors of *L*;
- otherwise, the union of each member of the crossproduct of the sets of nearest-defeasible-ancestors of the support-link-basis of *L* is inserted into the list of support-link-nearest-defeasible-ancestors of *L*.

The nearest-defeasible-ancestors of *N* is the union of the support-link-nearest-defeasible-ancestors for all support-links of *N*.

For example, consider the inference-graph of figure eight. Dashed arrows indicate defeasible inferences and solid arrows indicate deductive inferences. Accordingly, the list of nearest-defeasible-ancestors for *E* is {{*E*}}, and similarly for *F*–*L*. The list of nearest-defeasible-ancestors for *B* is {{*E*,*F*},{*G*,*H*}}, and that for *C* is {{*I*,*J*},{*K*,*L*}}. Finally, the list of nearest-defeasible-ancestors for *A* is {{*E*,*F*,*I*,*J*},{*E*,*F*,*K*,*L*},{*G*,*H*,*I*,*J*},{*G*,*H*,*K*,*L*}}.

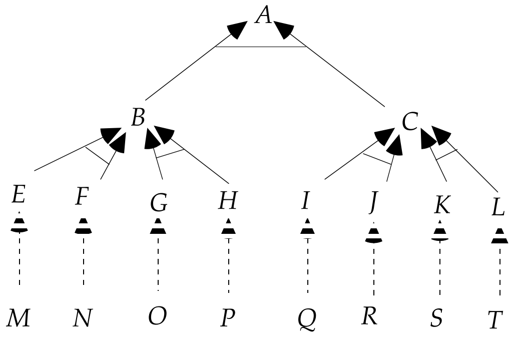

Figure Nearest-defeasible-ancestors.

We can also define:

N is a *deductive node* iff the list of its nearest-defeasible-ancestors is empty.

Epistemic cognition is performed by a flag-based reasoner. Such a reasoner consists of two semi-autonomous modules — a monotonic reasoner and a module that continually updates defeat-statuses as new inferences are made. The monotonic reasoner builds the *inference-graph*, and the defeat-status computation is then applied to the nodes of the *inference-graph*. The monotonic reasoner is an interest-driven reasoner, reasoning forwards from premises, perceptual inputs, and desires, and backwards from queries included in *ultimate-epistemic-interests*. The defeat-status computation is based upon the analysis of defeat-status proposed above. I will begin by discussing the defeat-status computation.

2. The Defeat-Status Computation

OSCAR employs an efficient algorithm for generating the status assignments used for computing defeat-status (i.e., the undefeated-degree-of-support). This results from noting that status assignments can be generated in a semi-recursive manner. For this purpose, it is convenient to redefine (partial) status assignments to be functions assigning either a number (the undefeated-degree-of-support) or "unassigned" (represented by NIL in the implementation):

> σ is a *partial status assignment* iff σ is a function assigning "unassigned" or real numbers between 0 and 1 to a subset of the nodes of an inference-graph and to the support-links of those nodes in such a way that:
> 1. σ assigns its maximal-degree-of-support to any initial node;
> 2. If σ assigns a numerical value α to a defeat-node for a support-link and assigns a numerical value less than or equal to α to some member of the link-basis, then σ assigns 0 to the link;
> 3. Otherwise, if σ assigns numerical values to every member of the link-basis of a link and every link-defeater for the link, σ assigns to the link the minimum of the strength of the link-rule and the numbers σ assigns to the members of the link-basis;
> 4. If every support-link of a node is assigned 0, the node is assigned 0;
> 5. If some support-link of a node is assigned a value greater than 0, the node is assigned the maximum of the values assigned to its support-links;
> 6. If every support-link of a node that is assigned a value is assigned 0, but some support-link of the node is not assigned a value, then the node is not assigned a value;
> 7. σ assigns "unassigned" to a support-link if it does not assign 0 to any member of the support-link-basis, it does not assign to any support-link-defeater a number greater than a number assigned to a member of the support-link-basis, and it either assigns "unassigned" to some member of the support-link-basis or to some support-link-defeater;
> 8. σ assigns "unassigned" to a node if some support-link for the node is assigned "unassigned", and all support-links not assigned "unassigned" are assigned 0.

> If σ and η are partial status assignments, σ is a *proper sub-assignment* of η iff σ and η have the same domain, σ ≠ η, and for every node or link to which σ assigns something other than "unassigned", η assigns the same value.

> σ is a *status assignment* iff σ is a partial status assignment and σ is not a proper sub-assignment of any other partial status assignment.

Given a partial assignment to a subset of the inference-nodes and support-links of the *inference-graph* and a set of stipulated assignments *arb* to some additional nodes, define the *assignment-closure* of *arb* and that assignment to be the minimal partial assignment containing *arb* and the given partial assignment, or NIL if there is none. This can be constructed by recursively applying the following rules until no further inference-nodes or support-links receive assignments or some inference-node or support-link receives inconsistent assignments:

- if some member of the support-link-defeaters has been assigned a numerical value greater than or equal to a numerical value assigned to some member of the link-basis or to the strength of the support-link-rule, assign 0 to the link;
- otherwise, if all support-link-defeaters and all members of the support-link-basis have been assigned values:
    - if some member of the support-link-basis of a support-link has been assigned "unassigned", assign "unassigned" to the link;

  - otherwise:
    - if some support-link-defeater for the link has been assigned “unassigned”, assign “unassigned” to the link;
    - otherwise, let *val* be the minimum of the strength of the support-link-rule and the numbers assigned to the members of the link-basis, and assign *val* to the link;
- if some support-link is assigned two different values, the assignment-closure is NIL;
- if every support-link of a node is assigned 0, the node is assigned 0;
- if some support-link of a node is assigned a numerical value greater than 0, and every support-link of the node is assigned some value, the node is assigned the maximum of the numerical values assigned to its support-links;
- if every support-link of a node is assigned some value, and every support-link of the node that is assigned a numerical value is assigned 0, but some support-link of the node is assigned “unassigned”, then the node is assigned “unassigned”.

Before discussing the algorithm for generating all maximal partial-assignments, consider a related (but simpler) algorithm for producing all total assignments, where the latter are defined as follows:

> A *total assignment* is a partial assignment that does not assign “unassigned” to any node.

The following algorithm will generate all total assignments:

```
• Let $\sigma_0$ be the assignment-closure of the partial assignment that assigns “undefeated” to all initial nodes, and let P-ass = {$\sigma_0$}.
• Let Ass = $\varnothing$.
• Repeat the following until no new assignments are generated:
    • If P-ass is empty, exit the loop.
    • Let $\sigma$ be the first member of P-ass:
        • Delete $\sigma$ from P-ass.
        • Let L be a support-link which has not been assigned a status but for which all members of the link-basis have been assigned non-zero values.
        • If there is no such node as L, insert $\sigma$ into Ass.
        • If there is such an L then:
            • Let S be the set of all assignments that result from extending $\sigma$ by assigning either 0 or the minimum value assigned to its link-basis to L.
            • Insert all non-empty assignment-closures of members of S into P-ass.
    • If Ass is unchanged, exit the loop.
• Return Ass as the set of assignments.
```

This algorithm generates all total assignments by trying to build them up recursively from below (ordering nodes in terms of the “inference-ancestor” relation). When this leaves the status of a link undetermined, the algorithm considers all possible ways of assigning statuses to that link, and later removes any combinations of such “aribtrary” assignments whose assignment-closures prove to be inconsistent. The general idea is that assignments are generated recursively insofar as possible, but when that is not possible a generate-and-test procedure is used.

To modify the above algorithm so that it will generate all maximal partial assignments, instead of just deleting inconsistent arbitrary assignments, we must look at proper sub-assignments of them. When such a proper sub-assignment has a consistent assignment-closure, and it is not a proper sub-assignment of any other consistent assignment, then it must be included among the maximal partial assignments. To manage this, the algorithm must keep track of which nodes have been assigned statuses arbitrarily in the course of constructing an assignment. Let σ_0 be the assignment-closure of the partial assignment that assigns “undefeated” to all initial nodes. Let us take an “annotated-assignment” to be a pair $\langle\sigma, A\rangle$ where A is an arbitrary assignment to some set of nodes and links, and σ is the assignment-closure of $\sigma_0 \cup A$. The algorithm constructs annotated-assignments.

COMPUTE-ASSIGNMENTS

```
• Let $\sigma_0$ be the assignment-closure of the partial assignment that assigns its degree of
  justification to any initial node, and let $P\text{-}ass = \{\langle\sigma_0,\varnothing\rangle\}$.
• Let $Ass = \varnothing$.
• Repeat the following until an exit instruction is encountered:
    • If P-ass is empty, exit the loop.
    • Let $\langle\sigma,A\rangle$ be the first member of P-ass:
        • Delete $\langle\sigma,A\rangle$ from P-ass:
        • Let L be a support-link that has not been assigned a status but for which
          all members of the link-basis have been assigned non-zero values.
        • If there is no such link as L, insert $\langle\sigma,A\rangle$ into Ass.
        • If there is such an L then:
            • Let Ass* be the set of all $A \cup X^*$ such that X* is an arbitrary assignment
              of either 0, “unassigned”, or the minimum value assigned to its link-
              basis to L.
            • Let S be the set of all maximal sub-assignments S* of members of
              Ass* such that the assignment-closure of $S^* \cup \sigma_0$ is non-empty.
            • For each member A of S:
                • If any member of P-ass is a sub-assignment of A, delete it from P-ass.
                • If any member of Ass is a sub-assignment of A, delete it from Ass.
                • If A is not a sub-assignment of any member of P-ass or Ass, insert A
                  into P-ass.
• Return as the set of assignments the set of all $\sigma$ such that for some A, $\langle\sigma,A\rangle$ is in
  Ass.
```

The correctness of this algorithm turns on the following observations:
(1) Every partial assignment can be generated as the assignment-closure of the assignment to the initial nodes and an arbitrary assignment to some otherwise undetermined links.
(2) If a partial assignment is inconsistent, so is every extension of it.
The algorithm makes use of (1) in the same way the previous algorithm did. In light of (2), in ruling out inconsistent assignments, we can shrink the search space by ruling out inconsistent sub-assignments and then only test for consistency the extensions of the remaining consistent sub-assignments.

Although this algorithm is much more efficient than a brute-force approach, it too encounters an efficiency problem. This algorithm constructs assignments for all nodes of the *inference-graph* simultaneously. To illustrate, consider the four-way collective defeat of figure ten. The preceding algorithm for computing assignments produces four assignments (one assigning “defeated” to each of nodes 5,6,7,8), in 0.07 seconds. But when we consider composite problems consisting of multiple instances of this single problem, the situation deteriorates rapidly:

Figure 10	0.07 sec 4 assignments	
2 instances of #10	1.20 sec 16 assignments	(increase by a factor of 17.14)
3 instances of #10	18.24 sec 64 assignments	(×260.57)
4 instances of #10	310.67 sec 256 assignments	(×4438.14)

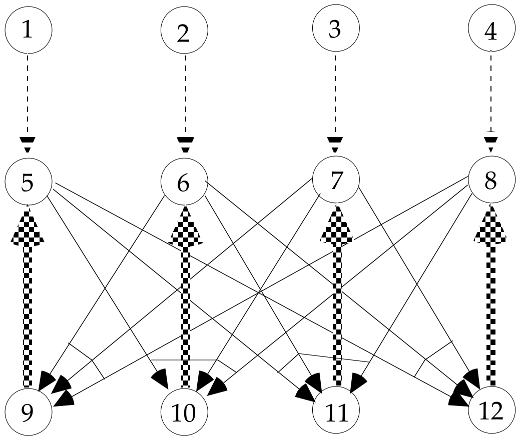

Figure 10. Four-way collective-defeat.

There is a better way of doing the computation. The different parts of the composite problem are really separate subproblems, and it should be possible to separate the computations, which would result in a linear increase in computational difficulty rather than an exponential one. Note that in realistic cases, this will be important, because the *inference-graph* of a real agent will generate a large base-assignment consisting of the assignment-closure of an assignment of "undefeated" to the initial nodes, and then a possibly large number of small independent problems consisting of collective defeat and kindred phenomena, each generating multiple extensions to the base assignment. It should be possible to handle each of these sets of multiple extensions independently, rather than lumping them all together.

Let us define the inference / defeat-ancestors of a support-link to be the set of inference-nodes and support-links that can be reached by working backwards via support-links and defeat-links. Precisely:

> *node* is an *inference/defeat-ancestor* of *link* iff either (1) *node* is a member of the link-basis of *link* or of a support-link that is an inference-defeat-ancestor of *link*, or (2) *node* is a defeater for *link* or for a support-link that is an inference-defeat-ancestor of *link*.
>
> A support-link *L* is an *inference/defeat-ancestor* of *link* iff *L* is a support-link for an inference-node that is an inference / defeat-ancestor of *link*.
>
> The inference / defeat-ancestors of a node are its support-links and their inference / defeat-ancestors.

In computing the defeat-status of a node or link, the only nodes and links that are relevant are its inference / defeat-ancestors. When we are presented with different subproblems embedded in a single inference-graph, we can divide the inference-graph into regions as in figure 11. Here *D* is the set of nodes and links assigned statuses by the base-assignment. The "triangle-sets" *A, B,* and *C* represent isolated subproblems. They are characterized by the fact that they are minimal sets *X* such that every inference / defeat-ancestor of a member of *X* is in either *X* or *D*. For the purpose of computing the defeat-statuses of the members of the triangle-sets, we can treat the $A \cup D$, $B \cup D$, and $C \cup D$ as separate inference-graphs. This will yield the same value for the defeat-status as a computation based upon the entire inference-graph.

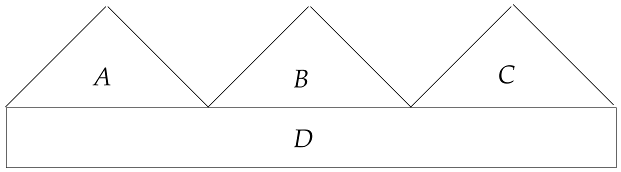

Figure 11. Triangle-sets.

The construction of triangle-sets can be made recursive, because there may be further triangle-sets that sprout above the existing ones. Notice, however, that higher-level triangle-sets must be relativized to assignments to the lower-level triangle-sets rather than to the lower-level triangle-sets themselves. This is because different assignments to a lower-level triangle-set may leave different sets of nodes unassigned and candidates for inclusion in a higher-level triangle-set.

A simple way to conceptualize this is in terms of an *assignment-tree*. The base of an assignment-tree is the base-assignment. That divides into a set of tree-nodes representing triangle-sets. Each of those tree-nodes gives rise to a set of assignments. Those assignments divide in turn into a new set of tree-nodes representing triangle-sets, and so on, as diagrammed in figure 12. This can be accommodated formally by taking an *assignment-tree* to be a data-structure consisting of an assignment and the list of resulting triangle-sets, and taking a *triangle-set* to be a data-structure consisting of a set of inference-nodes and support-links (its domain) and the list of assignment-trees generated by the assignment-closures of maximal-consistent assignments to the inference-nodes and support-links in the triangle-sets. For the purpose of evaluating the defeat-statuses of the nodes of the *inference-graph*, we can construct the assignment-tree *global-assignment-tree* generated by the base-assignment. Then a n inference-node *node* is undefeated iff the *inference-graph* contains an argument for *node* which is such that no assignment in *global-assignment-tree* assigns 0 or "unassigned" to any nodes or links in the argument

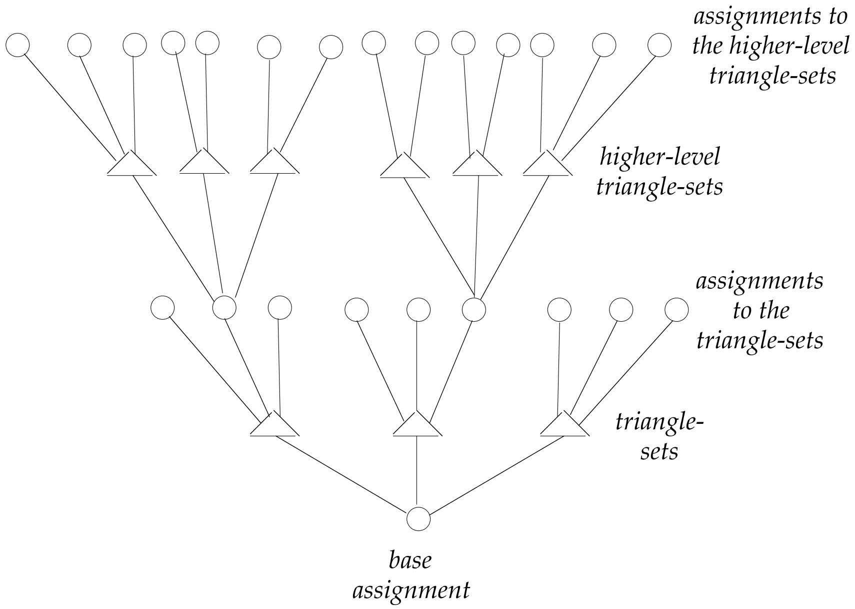

Figure 12. An assignment-tree.

The complete assignments operative in a triangle-set will be the result of appending all of the assignments on the branch of the assignment-tree leading from the base assignment to the triangle-set, and then appending to that the assignments to the triangle-set. The list of assignments on the branch leading to the triangle-set, beginning with the base assignment, will constitute the *ancestor-assignments* of the triangle-set. These will consist of the assignment to the parent-tree, and the assignment to its parent-tree, and so on. These are computed recursively by working backwards through the tree. The assignment-closure of a new partial assignment must accordingly be understood as relative to an assignment-tree. A node or link *has a defeat-status* in a tree iff it is assigned a status either by the assignment of the tree or one of its ancestor assignments.

Given an assignment-tree, the following returns a list of triangle-domains (taken to be sets of support-links), each ordered so that the link generating it is the first member:

```
COMPUTE-TRIANGLE-DOMAINS tree
    • Let X be the set of all consequent-links of inference-nodes in the domain of the assignment of tree that do not have defeat-statuses in tree but whose support-link-bases consist of nodes that do have defeat-statuses in tree.
    • If X is empty, return the empty set.
    • If X is nonempty, let triangle-domains be empty. Then for each link in X:
        • Let triangle be the set of inference/defeat-ancestors of link that do not have defeat-statuses in tree.
        • If link is a member of triangle, and there is no triangle* already in triangle-domains such that the first member of triangle* is a member of triangle, then insert triangle into triangle-domains, ordering the members of triangle so that link is the first member.
    • Return the list triangle-domains.
```

global-assignment-tree is then computed by applying the following algorithm, with no arguments. Each time a triangle-set is constructed and assignments to it are computed, COMPUTE-ASSIGNMENT-TREE is called recursively with arguments listing the assignment to the domain of the triangle-set and the parent-tree.

```
COMPUTE-ASSIGNMENT-TREE optional-arguments: assignment parent-tree
    • If no values are provided for the optional-arguments:
        • Define assignment to be the assignment-closure of the partial assignment that assigns “undefeated” to all initial nodes.
    • Let tree be a new assignment-tree whose assignment is assignment and whose parent-tree is parent-tree (if the latter is provided).
    • Let triangles be the list of triangle-domains computed by applying COMPUTE-TRIANGLE-DOMAINS to tree.
    • Set the list of triangle-sets of tree to be the result of applying COMPUTE-TRIANGLE-SET to each member of triangles together with tree.
```

```
COMPUTE-TRIANGLE-SET triangle-domain tree
    • Let S be the set of all maximal consistent assignments to triangle-domain relative to tree, computed by applying COMPUTE-ASSIGNMENTS-TO-DOMAIN to tree and triangle-domain,
    • Construct triangle-set to be the triangle-set whose domain is triangle-domain and whose list of assignment-trees are generated by applying COMPUTE-ASSIGNMENT-TREE to each member ass of S and tree.
```

```
COMPUTE-ASSIGNMENTS-TO-DOMAIN tree triangle
    • Let link be the first member of triangle.
    • Where min-val is the minimum of the reason-strength of the support-link-rule of link and the values assigned to the members of the support-link-basis of link, let Ass* be the set {{⟨link,0⟩},{⟨link,min-val⟩}}.
• Let P-ass be the set of all pairs $\langle closure,A\rangle$ where $A$ is in $Ass^*$, closure is the
  assignment-closure of $A$ relative to tree, and closure is nonempty.
• If P-ass is empty, return the list whose only member is the assignment-closure of
  $\{\langle link,$"unassigned"$\rangle\}$ relative to tree.
• If P-ass is nonempty:
    • Let $Ass = \varnothing$.
    • Repeat the following until an exit instruction is encountered:
        • If P-ass is empty, exit the loop.
        • Let $\langle \sigma,A\rangle$ be the first member of P-ass:
            • Delete $\langle \sigma,A\rangle$ from P-ass:
            • Let $L$ be a link in triangle that has not been assigned a status by $\sigma$, but
              for which all members of the support-link-basis have been assigned
              non-zero numerical values.
            • If there is no such link as $L$, insert $\langle \sigma,A\rangle$ into Ass.
            • If there is such an $L$ then:
                • Let $Ass^*$ be the set of all $A \cup X^*$ such that $X^*$ is an arbitrary assignment
                  to $L$ of 0 or of the minimum of the reason-strength of its support-link-
                  rule and the values assigned to the members of its support-link-basis.
                • Let $S$ be the set of all maximal sub-assignments $S^*$ of members of $Ass^*$
                  such that the assignment-closure of $S^*$ relative to tree is non-empty.
                • For each member $A$ of $S^*$, let closure be its assignment-closure relative
                  to tree:
                    • If any member of P-ass is a sub-assignment of closure, delete it from
                      P-ass.
                    • If any member of Ass is a sub-assignment of closure, delete it from
                      Ass.
                    • If $A$ is not a sub-assignment of any member of P-ass or Ass, insert
                      $\langle closure,A\rangle$ into P-ass.
    • Return as the set of assignments the set of all $\sigma$ such that for some $A$, $\langle \sigma,A\rangle$ is
      in Ass.
```

The difference between the performance of the algorithms is illustrated dramatically by considering composites of figure ten again. For a single instance of that problem, the original assignments-algorithm is the same speed as the assignment-tree-algorithm. But when we consider composite problems, the comparison changes dramatically:

	assignments-algorithm		**assignment-tree-algorithm**	
Figure 10	0.07 sec 4 assignments		0.07 sec 4 branches	
2 instances of #10	1.20 sec 16 assignments	(×17.14)	0.15 sec 8 branches	(×2.14)
3 instances of #10	8.24 sec 64 assignments	(×260.57)	0.25 sec 12 branches	(×3.57)
4 instances of #10	310.67 sec 256 assignments	(×4438.14)	0.35 sec 16 branches	(×5.0)

The difference is attributable to the fact that n copies of a simple problem that produces k assignments and an assignment-tree with k branches will produce n^k assignments but will produce an assignment-tree with only $n \cdot k$ branches. The times required to run the algorithms will be roughly proportion to the number of assignments and branches produced. The proportions are not exact because partial branches and assignments may be produced and later discovered to be inconsistent and so pared from the tree or dropped from the list of assignments. A set of test problems has been produced which encode the inference-graphs of figures 2, 3, 4, 5, 7, 8, 9, 10, 14, 18 from chapter three of *Cognitive Carpentry*, and composites thereof. The results are as follows:

	assignments-algorithm	**assignment-tree-algorithm**
Problem #2	0.01 sec 1 assignment	0.01 sec 1 branch
Problem #3	0.02 sec 2 assignments	0.01 sec 2 branches
Problem #4	0.03 sec 2 assignments	0.04 sec 2 branches
Problem #5	0.04 sec 2 assignments	0.03 sec 2 branches
Problem #7	0.04 sec 2 assignments	0.03 sec 2 branches
Problem #8	0.05 sec 3 assignments	0.05 sec 3 branches
Problem #9	0.01 sec 1 assignment	0.01 sec 1 branch
Problem #10	0.01 sec 1 assignment	0.01 sec 1 branch
Problem #11	0.01 sec 1 assignment	0.01 sec 1 branch
Problem #12	0.02 sec 2 assignments	0.02 sec 2 branches
Problem #14	0.02 sec 1 assignment	0.02 sec 1 branch
Problem #18	0.11 sec 1 assignment	0.07 sec 1 branch
Problem #19 *composite of 18, 18*	0.53 sec 1 assignment	0.18 sec 2 branches
Problem #20 *composite of 18, 18, 18*	1.71 sec 1 assignment	0.30 sec 3 branches
Problem #21 *composite of 18, 18, 18, 18*	4.63 sec 1 assignment	0.45 sec 4 branches
Problem #22 *composite of 2, 3, 4*	0.29 sec 4 assignments	0.06 sec 4 branches

Problem #23	1.39 sec	0.11 sec
composite of 2, 3, 4, 5	8 assignments	6 branches
Problem #24	27.40 sec	0.22 sec
composite of 2, 3, 4, 5, 7, 8	48 assignments	11 branches
Problem #25	34.92 sec	0.26 sec
composite of 2, 3, 4, 5, 7, 8,	48 assignments	13 branches
9, 10		
Problem #26	62.10 sec	0.47 sec
composite of 2, 3, 4, 5, 7, 8,	48 assignments	15 branches
9, 10, 14, 18		

We have an efficient algorithm for computing defeat-statuses for an *inference-graph* that has already been constructed. However, that is an artificial problem. A rational agent must construct its *inference-graph* one support-link at a time, and update the defeat-statuses with the addition of each link. The result of updating the assignment-tree with the addition of each link should be the same as the result of computing the assignment-tree as above for the completed *inference-graph*.

One of the virtues of the assignment-tree algorithm is that it leads to an efficient procedure for updating the defeat-statuses of the nodes of the *inference-graph* in response to adding new support-links. When a link is added to the *inference-graph*, the only pre-existing nodes and links that can be affected are the inference/defeat-descendants of the support-link-target of the new link. The latter concept is defined as follows:

> The *inference/defeat-descendants* of a support-link consist of its target and all of the inference/defeat-descendants of its target; the *inference/defeat-descendants* of an inference-node consist of (1) all the consequent-links of the node and all the inference/defeat-descendants of those consequent-links, and (2) all the node-defeatees of the node and all the inference/defeat-descendants of those node-defeatees.

When we add new links, the changes to the inference-graph are confined to the new links and their inference/defeat-descendants. Call these the *affected-nodes* and *affected-links*. If the affected-nodes have no defeatees, we can just compute the assignments to the affected-nodes and affected-links at any point in the assignment tree at which they become computable, and add those assignments to the assignment-tree. Otherwise, we must update *global-assignment-tree*.

Updating a tree in response to adding a support-link is performed only if updating the lower-order trees has not changed anything. In updating a tree, it must first be determined whether its assignment assigns anything to affected-nodes or affected-links. If not, it is determined whether the newly added link has a computable defeat status. If it does the assignment of the tree is redefined to be the result of adding that to the tree's assignment and forming the assignment-closure. Then the triangle-sets of the tree are recomputed. If the new link does not have a computable status, the triangle-sets of the tree are updated. If these are unchanged, the assignment-trees of the triangle-sets are updated, and so on.

In updating a triangle-set, a check is made to determine whether any of the new links have computable statuses in the tree or whether any other affected-nodes or affected-links already occur in its domain. If neither of these conditions is satisfied, its assignment-trees are updated. But if some members of its domain are affected, the domain must be recomputed to be the set of inference/defeat-ancestors of the original domain. The new domain will contain some *affected-nodes*. Then the triangle-set must be recomputed from scratch and the old one replaced by the new one.

In updating an assignment-tree, let *ass0* be the assignment to the unaffected-nodes and unaffected-links that are in the domain of the original assignment, and define *closure* to be the closure of *ass0*. We have the following possibilities:

If *closure* were inconsistent, then the *new-links* would have to render some of the arbitrary assignments inconsistent, in which case the domain of the enclosing triangle-set would contain affected nodes or links and so the recursion would not lead to this assignment-tree being updated. Hence we can assume that *closure* is consistent.

If *closure* makes the same assignments as the original assignment of the assignment-tree, then update the tree's triangle-sets. But if *closure* differs from the original assignment, redefine the assignment of the assignment-tree to be *closure*, and recompute the triangle-sets of the assignment-tree. If a triangle-domain is unchanged, then update the trees of that triangle-set. For triangle-domains that are different from the original ones, the triangle-sets must be computed from scratch.

Whenever we change the assignment of an assignment-tree, we must change the list of defeating-assignment-trees for support-links whose assignments change, and recompute the undefeated-degrees-of-support of the support-link-targets and their inference-defeat descendants to reflect that change:

These observations produce the following pseudocode:

```
UPDATE-DEFEAT-STATUSES
    Let affected-nodes and affected-links consist of new-links and the inference/defeat-descendants if members of new-links. If every affected-node has an empty list of node-defeatees, compute the status-assignments for the affected-nodes and affected-links in the various subtrees of global-assignment-tree in which those status-assignments become computable, and insert the computed values into the subtrees. Otherwise, apply UPDATE-ASSIGNMENT-TREE to new-links, global-assignment-tree, affected-nodes and affected-links.
```

```
UPDATE-ASSIGNMENT-TREE links tree affected-nodes affected-links
    • Let assignment be the assignment of tree. If assignment does not assign statuses to any members of affected-nodes or affected-links other than members of links:
        • If some members of links: have computable numerical defeat-statuses relative to tree, let closure to be the assignment-closure of that assignment to link relative to tree. Redefine the assignment of tree to be the result of adding closure to assignment, and apply RECOMPUTE-TRIANGLE-SETS to links, tree affected-nodes and affected-links.
        • If no members of links have computable numerical defeat-statuses val relative to tree, apply UPDATE-TRIANGLE-SET to links, triangle-set, affected-nodes, affected-links, and tree for each triangle-set of tree.
    • Otherwise, let ass0 be the assignment to the unaffected-nodes and unaffected-links that are in the domain of assignment. Let closure be the assignment-closure of ass0 for the affected-nodes.
        • If closure agrees with assignment, apply UPDATE-TRIANGLE-SET to links, triangle-set, affected-nodes, affected-links, and tree for each triangle-set of tree.
        • Otherwise, redefine the assignment of the assignment-tree to be closure, and apply RECOMPUTE-TRIANGLE-SETS to links, tree affected-nodes and affected-links.
```

```
RECOMPUTE-TRIANGLE-SETS links tree affected-nodes affected-links
    • Recompute the triangle-domains of tree.
    • If a triangle-domain is unchanged, then apply UPDATE-TRIANGLE-SET to it (together with links, affected-nodes, affected-links, and tree).
    • For triangle-domains that are different from the original ones, replace the triangle-set by the result of applying COMPUTE-TRIANGLE-SET to the new domain and tree.
```

```
UPDATE-TRIANGLE-SET links triangle-set affected-nodes affected-links tree
    If the domain of triangle-set does not contain any affected-links, and it does not contain any support-links having members of affected-nodes among their support-link-bases or support-link-defeaters, apply UPDATE-ASSIGNMENT-TREE to the assignment-trees of
```

triangle-set (relative to *links*, *affected-nodes*, and *affected-links*). Otherwise:

- Recompute the domain of *triangle-set* to be the set of inference/defeat-ancestors of the original domain.
- Replace *triangle-set* in *tree* by a new triangle-set constructed by applying COMPUTE-TRIANGLE-SET to the new domain and the result of adding the assignment of *tree* to its ancestor assignments.

Updating defeat-statuses is initiated whenever a new support-link is constructed (by DRAW-CONCLUSION or DRAW-REDUCTIO-CONCLUSION, as discussed below), by executing UPDATE-BELIEFS:

UPDATE-BELIEFS

- UPDATE-DEFEAT-STATUSES, letting *altered-nodes* be the inference-nodes that are inference/defeat-descendants of the members of *new-links*.
- COMPUTE-UNDEFEATED-DEGREES-OF-SUPPORT, letting *new-beliefs* be the list of nodes whose undefeated-degrees-of-support (i.e., defeat-statuses) increase as a result of this computation, and *new-retractions* be the list of nodes whose undefeated-degrees-of-support decrease as a result of this computation.
- COMPUTE-INTEREST-GRAPH-DEFEAT-STATUSES using *new-beliefs and new-retractions*, letting *altered-interests* be the *interest-graph* nodes whose defeat-statuses change.
- For each support-link in *new-links*, if its support-link-target has no support-link not in *new-links*, insert the support-link-target into the *inference-queue*.
- For each member *S* of *new-beliefs*, APPLY-OPTATIVE-DISPOSITIONS-TO *S* and APPLY-Q&I-MODULES-TO *S*.
- DISCHARGE-ULTIMATE-EPISTEMIC-INTERESTS *new-beliefs new-retractions*, letting *altered-queries* be the list of queries whose query-answered-flags change value as a result.
- RECOMPUTE-PRIORITIES using *new-beliefs, new-retractions*, *altered-interests* and *altered-queries*, and reorder the *inference-queue*.

As indicated above, COMPUTE-UNDEFEATED-DEGREES-OF-SUPPORT proceeds recursively. Reformulating the previously described algorithm in terms of defeating-assignment-trees, we have:

COMPUTE-UNDEFEATED-DEGREES-OF-SUPPORT

- Find all support-links that have lost or acquired defeating-assignment-trees. This is the set of altered-links.
- For each altered-node, set its old-undefeated-degree-of-support to be its current undefeated-degree-of-support.
- Let N_0 be the set of support-link-targets for the altered-links, and let *N* be the set of inference-descendants of N_0. For each member of *N*, set its undefeated-degree-of-support in terms of those of its support-links that do not have defeating-assignment-trees and whose bases are disjoint from *N* (set it to 0 if there are none).
- For each node in *N*, if it has a nonzero undefeated-degree-of-support, then for each of its consequent-links *L* that have no defeating-assignment-trees, compute a undefeated-degree-of-support for the target of *L* as the minimum of the support-link-strength of *L* and the undefeated-degrees-of-support of its basis. If that is greater than the existing undefeated-degree-of-support, recurse that change through the node-descendants of the target using altered-links.

3. Overview of the Monotonic Reasoner

The monotonic reasoner is interest-driven, as described above. Reasoning forwards builds the *inference-graph*, and reasoning backwards builds the *interest-graph*. When the two graphs meet, interests are discharged, with the result that sequents that are of interest (as recorded in the *interest-graph*) are concluded and incorporated into the *inference-graph*. Queries posed by practical cognition form termini in the *interest-graph*. When such queries are answered, the agent does something with the answers. What the agent does is determined by the purpose of the query. This will be accomplished by storing instructions for what to do with an answer along with the query in *ultimate-epistemic-interests*. Sometimes these instructions will produce new queries for insertion into *ultimate-epistemic-interests* and thereby initiate new backwards reasoning.

Interests comes in degrees, and so do the degrees-of-support for members of the *inference-graph*. In accordance with the weakest link principle, the degree-of-support for a support-link is the minimum of (1) the strength of the reason and (2) the maximal-degrees-of-support for the members of the link-basis. The maximal-degree-of-support for an inference-node is normally the maximum of the degrees of support for its support-links. Degrees of interest originate with the permanent members of *ultimate-epistemic-interests*. When new members of *ultimate-epistemic-interests* are produced, the instructions for producing them must also set their degrees of interest. Reasoning backwards from *ultimate-epistemic-interests* preserves the degree-of-interest. In general, if a sequent in the *interest-graph* is of interest for more than one reason (i.e., there are branches of the *interest-graph* leading from it to more than one query in *ultimate-epistemic-interests*), the degree-of-interest in the sequent is the minimum of the degrees of interest in the queries from which it derives. This indicates the minimal undefeated-degree-of-support an answer must have in order to satisfy (one source of) the interest.

The reasoning recorded in the *inference-graph* is defeasible. A necessary condition for one node to defeat another is that the maximal-degree-of-support for the defeating node be at least as great as that for the defeated node. Defeat-statuses are the same as undefeated-degrees-of-support, and are computed as described in section one. A query in *ultimate-epistemic-interests* is only deemed to have been answered if the answer found has an undefeated-degree-of-support at least as great as the degree-of-interest in the query.

The basic control structure for these inference operations is the *inference-queue*. This is a queue of inferences (either backwards or forwards) waiting to be performed. The details of the ordering can be changed without changing the OSCAR architecture, but it is presumed that the ordering is sensitive to degrees of support and degrees of interest. This will be discussed in more detail below.

The basic data-structures used by epistemic cognition will be sequents, backwards and forwards reasons, the *inference-graph*, the lists of *conclusions* and *processed-conclusions*, the *interest-graph*, the *inference-queue*, and the list of *ultimate-epistemic-interests*. The structure of the *inference-graph* was described above. The others are as follows:

Formulas

OSCAR uses an open-ended syntax. That is, no definition of the set of formulas is presupposed, but it is presupposed that formulas can be combined and constructed from one another by using operations like conjunction, negation, or quantification. The details of this are contained in the file *Syntax.lsp*.

OSCAR assumes the following logical constants: ~, &, v, ->, <->, @, all, some, =. Note that the logical constants are always quoted. Note also that v is the lower-case letter v, so that cannot be used as a variable for quantification. @ is used for the undercutting symbol ⊗, which may not be readily available in the character sets used by LISP.

Formulas are lists or trees of symbols. For example, the following is a formula:

```
(((all x) ((P x) -> (~ (Q x)))) v ((some y) ((P y) & (R y))))
```

This illustrates several points. If P is a formula, its negation is (~ P). Quantifiers have the form (Q var) where Q is either all or some. Given a quantifier (Q var) and a formula P, ((Q var) P) is a quantified formula. We cannot just append ~ or a quantifier to the front of a formula, because that is not a LISP list. Instead, we must enclose the result in parentheses. Similarly, if F is a predicate, formulas are constructed from it by constructing lists of the form (F x1 … xn). We cannot use more standard notation like F(x1 … xn"), because that is not a LISP list.

Formulas are treated in this way because they are easily manipulated by LISP functions. However, it is difficult for human beings to read these expressions. Accordingly, OSCAR distinguishes between formulas and *pretty-formulas*. Pretty-formulas are strings constructed from formulas by the function PRETTY, and look more like standard logical notation. For instance, the pretty-formula corresponding to the above formula is:

```
"[(all x)((P x) -> ~(Q x)) v (some y)((P y) & (R y))]"
```

The function REFORM converts a pretty-formula into a formula. Pretty formulas append negations and quantifiers to subformulas rather than listing them jointly. Pretty-formulas also allow the use of brackets in place of parentheses for enhanced readability.

It is possible to use non-standard syntax in pretty-formulas by giving OSCAR explicit information about how to translate them into formulas. This is done by putting entries into the **reform-list**. The following is an example of a specification of the **reform-list**:

```
(let ((P (gensym)) (Q (gensym)) (x (gensym)) (y (gensym)) (A (gensym)) (z (gensym)) (op (gensym)))
 (setf *reform-list* nil)
 (dolist
   (pair
    (list
     `((,P throughout (,op ,x ,y)) (throughout ,P (,op ,x ,y)) (,P ,op ,x ,y))
     `((,P at ,x) (throughout ,P (closed ,x ,x)) (,P ,x))
     `((,P now) (at ,P now) (,P))
     `(((it appears to me that ,Q) at ,x) (it-appears-to-me-that ,Q (closed ,x ,x)) (,Q ,x))
     `((the probability of ,P given ,Q) (the-probability-of ,P ,Q) (,P ,Q))
     `((I  have a percept with content ,Q) (I-have-a-percept-with-content ,Q) (,Q))
     `((,x < ,y) (< ,x ,y) (,x ,y))
     `((,x <= ,y) (<= ,x ,y) (,x ,y))
     `((,x = ,y) (= ,x ,y) (,x ,y))
     `((,x + ,y) (+ ,x ,y) (,x ,y))
     `((,x * ,y) (* ,x ,y) (,x ,y))
     `((,x expt ,y) (expt ,x ,y) (,x ,y))
     `((,x - ,y) (- ,x ,y) (,x ,y))
     `((,x is a reliable informant) (reliable-informant ,x) (,x))
     `((,x reports that ,P) (reports-that ,x ,P) (,x ,P))
     `((the color of ,x is ,y) (color-of ,x ,y) (,x ,y))
     `((,x and ,y collide) (collide ,x ,y) (,x ,y))
     `((the position of ,x is (,y ,z)) (position-of ,x ,y ,z) (,x ,y ,z))
     `((the velocity of ,x is (,y ,z)) (velocity-of ,x ,y ,z) (,x ,y ,z))
     `((,x is dead) (dead ,x) (,x))
     `((,x is alive) (alive ,x) (,x))
     `((,x is a dimensionless billiard ball) (dimensionless-billiard-ball ,x) (,x))
     ))
   (pushnew pair *reform-list* :test 'equal)))
```

The entries in the **reform-list** are triples. The first member of a triple is the expression that REFORM returns when applied to the pretty-formula. The second member is formula we want it to return instead. The third member is the list of variables used in pattern-matching.

Sequents

Sequents will be ordered pairs ⟨*X*,*P*⟩ where *X* is the *sequent-supposition* and *P* is the *sequent-formula*. I will generally write the sequent in the form "*P*/*X*", which is often easier to read. Let us say that a sequent is *concluded* iff it is the node-sequent of some node of the *inference-graph* of kind "inference". A sequent can be concluded without being justified, because the node supporting it may be defeated. A sequent is *justified to* *degree* δ iff it is the node-sequent of a node of the *inference-graph* of kind "inference" whose undefeated-degree-of-support is δ. By the principle of foreign adoptions, if a sequent has been concluded, it can automatically be used in inferences involving more inclusive suppositions, so it is also convenient to define a sequent Q/Y to *subsume* a sequent P/X iff $P = Q$ and $Y \subseteq X$, and then say that a sequent is *validated* iff it is subsumed by some sequent that is concluded. In accordance with the principle of foreign adoptions, inferences will be from validated sequents rather than just from concluded sequents. I will say that a node of the *inference-graph* *supports* a sequent iff it is the node-sequent of that node, and the node *validates* a sequent P/X iff for some Y such that $Y \subseteq X$, P/Y is the node-sequent of that node.

Conclusions and Processed-Conclusions

The set of inference-nodes will be stored in the list of *conclusions*, and inference rules will access that list rather than looking directly at the *inference-graph*. One complication arises for the appeal to *conclusions*. Consider a forwards-reason having multiple premises. For instance, suppose $\{P,Q\}$ is a reason for R. Suppose P and Q are both on the *inference-queue*. If the reasoner retrieves P first, it will search the list of *conclusions* for Q, find it, and then make the inference to R. When it later retrieves Q from the *inference-queue*, if it searches the list of *conclusions* for P, it will find it, but we do not want the reasoner to repeat the inference to R. To avoid this, when the reasoner retrieves one premise from the *inference-queue*, in its search for the other premises, we only want it to consider conclusions that have already been retrieved from the *inference-queue*. To achieve this, the reasoner will keep a separate list of *processed-conclusions*. A conclusion will be inserted into this list only when it is retrieved from the *inference-queue*.

In the interest of efficient search, *conclusions* and *processed-conclusions* are lists of indexed data structures rather than simple lists. They are characterized by lists of pairs ⟨*formula,conclusion-list*⟩ where *formula* is the sequent-formula of a node-sequent, and *conclusion-list* is the list of all conclusions whose node-sequents have that sequent-formula. It is more efficient, however, to replace the conclusion-lists by structures called "c-lists", which encode the following information:

- c-list-formula
- corresponding-i-list
- c-list-nodes
- c-list-processed-nodes
- link-defeatees — the list of interest-links (see below) defeated by the conclusions in the c-list.

The c-list *for* a formula is the unique c-list having that formula as its c-list-formula. The advantage of this approach is that in LISP we can keep a record in a node of the c-list in which the node is included, and as the node is updated by adding support-links, no changes have to be made to the c-list. This makes it easy to move back and forth between nodes and their c-lists.

The most costly operations performed in reasoning are searches through lists of inference-nodes and interests. Great effort has gone into minimizing search in OSCAR. The use of c-lists is one mechanism for doing this. When searching for an inference-node supporting a sequent, we first search for the c-list-for the sequent-formula, and then we need only search the list of c-list-nodes of that c-list.

Another mechanism for minimizing search involves storing interests in similar structures called "i-lists", and to minimize the searches involved in discharging interest, whenever a c-list or i-list is constructed, the "corresponding" i-list and c-list involving the same formula is found (if it exists) and recorded in the slots "corresponding-i-list" and "corresponding-c-list". Once the corresponding i-list and c-list are found, the search never has to be repeated.

Rather than storing c-lists and i-lists in lists, they are actually stored in a discrimination-net. This is a form of indexed database that can be searched more efficiently than a list. The details of this are described in chapter six.

Ultimate-epistemic-interests

Querying epistemic cognition is done by inserting a query into *ultimate-epistemic-interests*. A query will be a data-structure encoding the following:

- the query-formula — a formula expressing the question asked.
- query-strength — the degree-of-interest, a number between 0 and 1.
- query-queue-node — if the query is currently on the *inference-queue*, this lists the queue-node enqueing it.
- deductive-query — T if the objective of the query is to determine whether the query-formula is deductively provable; NIL otherwise.
- the positive-query-instruction — an instruction for what to do with an answer.
- the negative-query-instruction — an instruction for what to do if an answer is withdrawn (explained below).
- query-answers — the list of all inference-nodes whose node-sequents constitute answers to the query.
- answered? — the "query-answered-flag"; this is T if any of the query-answers is justified to a degree greater than or equal to the degree-of-interest, NIL otherwise.
- query-interest — the interest for the query-formula. Interests are discussed below.

There are two kinds of queries. Sometimes epistemic cognition is queried about whether something is true. In this case, the query-formula is a closed formula, and the query is a *definite query*. Sometimes epistemic cognition is asked to find something (anything) of a certain general sort (e.g., a plan). In this case, the query-formula contains free variables. For example, in instructing epistemic cognition to try to find a plan satisfying some condition *C*, the query-formula will be ⌜*x* is a plan satisfying condition *C*⌝, where *x* is a free variable. Such a query is an *indefinite query*. In the implementation, the query-formula for an indefinite query is constructed using a special quantifier: ⌜(?*x*)φ⌝. An answer to a definite query will be an inference-node supporting the query-formula, and an answer to an indefinite query will be an inference-node supporting an instance of the query-formula.

It can happen that an answer to a query is found, and the positive-query-instruction executed, but the answer is later retracted. In some cases there may be nothing that can be done about that. For example, if the answer led to the adoption and execution of a plan, there may be no way to take it back. If there is, that can be done by adopting another plan, using the same planning procedures. In special cases, there may be a predictable way of repairing or partially repairing the results of the mistake, and that can be built into the instructions accompanying the query. To accommodate this, we allow the query to include instructions both for what to do when an answer is found — the positive-query-instruction — and for what to do if an answer is retracted — the negative-query-instruction. Either or both of these instructions can be omitted.

Some queries will be built into *ultimate-epistemic-interests* from the start, without having to be inserted by practical cognition. These will comprise the list of *permanent-ultimate-epistemic-interests*. Among the *permanent-ultimate-epistemic-interests* will be an interest in finding suitable goals. Here the interest is in finding situation types having positive expected likability. The instructions will be that when such a situation type is found, interest in finding minimally acceptable plans for achieving it is inserted into *ultimate-epistemic-interests*, and if the reasoner subsequently retracts the belief that a situation type has a positive expected likability, then the latter interest is withdrawn from *ultimate-epistemic-interests*.

In the doxastic implementation of practical reasoning in the OSCAR architecture, the only way new ultimate epistemic interests are adopted is in response to instructions from queries already in *ultimate-epistemic-interests* (as in figure one of chapter two). A general constraint on these instructions must be that whenever a new query is inserted into *ultimate-epistemic-interests*, this is done using the operation ADOPT-ULTIMATE-INTEREST. Let us say that a sequent is *in interest* iff it is the node-sequent of some *interest-graph-node* (see below). Then define:

ADOPT-ULTIMATE-INTEREST *query*

- Insert *query* into *ultimate-epistemic-interests*.

- For each inference-node *C*, if *C* answers *query*, insert *C* into the list of query-answers for *query*.
- If some member of the the list of query-answers for *query* is justified to a degree greater than or equal to the degree-of-interest in *query*:
  - Set *answered?* for *query* to T.
  - For each adequately justified inference-node *C* in the list of query-answers for *query*, perform the positive-query-instruction of *query* on the sequent-formula of the node-sequent of *C*.
- Otherwise, set *answered?* for *query* to NIL.
- If the query-formula of *query* is not already in interest, insert *query* into the *inference-queue*.

The details of inserting a query into the *inference-queue* will be discussed below.

The interest-graph

In the most general case, a backwards reason *R* has a list of forwards-premises, a list of backwards-premises, a desired-conclusion, and a list of variables to be used in pattern matching. If *R* is applicable to a desired-conclusion *S*, the reasoner matches *S* against the desired-conclusion, obtaining instantiations for some of the variables and thereby generating an *interest-scheme* ⟨*X*,*Y*,*S*⟩. The reasoner then looks for ways of instantiating the forwards-premise schemes *X* so that they are validated by inference-nodes that have already been drawn and are supported to the requisite undefeated-degree-of-support. A constraint on backwards reasons is that all the variables occur in the forwards premises and the conclusion, so once instances of the forwards-premises and the conclusion have been obtained, that uniquely determines the corresponding instance of the backwards-premises. Thus the reasoner obtains an instance ⟨*X**,*Y**,*S*⟩ of *R* where the members of *X** are validated to the requisite undefeated-degree-of-support. The reasoner then reasons backwards and adopts interest in the members of *Y**.

Subsequently, when a new conclusion *C* is drawn, DISCHARGE-INTEREST-SCHEMES checks the interest-scheme ⟨*X*,*Y*,*S*⟩ to see whether *C* can be used to instantiate some member of *X*, and if so whether the instantiation can be extended so that all members of *X* are instantiated to validated sequents. This yields a new instance ⟨*X***,*Y***,*S*⟩ of *R* where, once again, the forwards-premises *X*** are validated to the requisite undefeated-degree-of-support, and then the reasoner reasons backwards and adopts interest in the members of *Y***.

To implement the above reasoning, the interest-scheme ⟨*X*,*Y*,*S*⟩ must be stored somewhere so that it can be accessed by DISCHARGE-INTEREST-SCHEMES. Along with it must be stored the list of variables remaining to be instantiated, and the undefeated-degree-of-support required for validating instances. Accordingly, I will take an interest-scheme to be a data structure encoding the following information:

- the instantiated-reason.
- the instance-function — discussed below.
- the forwards-basis — a list of formulas.
- the backwards-basis — a list of formulas.
- the instance-consequent — the sequent of interest.
- the target-interest — the *interest-graph* node recording interest in the instance-conclusion.
- the instance-variables — the set of variables to be used in finding instances of the forwards-basis by pattern matching.
- the instance-strength — the degree of interest in the consequent, a number between 0 and 1.
- instance-supposition — the sequent-supposition of the instance-consequent.
- deductive-instance — T if the interest is in finding a deductive proof of the instance-consequent; NIL otherwise.
- scheme-discharge — if the inference-rule involves discharging a supposition (e.g., conditionalization), this is the supposition to be discharged.
- scheme-generating-node — if the interest scheme is generated by a supposition

(as in conditionalization or reductio-ad-absurdum), this is the node encoding the supposition.

I will abbreviate reference to interest-schemes by writing the septuple consisting of the values of the first seven slots.

In most backwards-reasons, the list of forwards-premises is empty (i.e., they are "simple" backwards reasons). In that case, instantiating the reason-conclusions uniquely determines the instantiation of the backwards-premises, and so no interest-schemes need be stored. The reasoner immediately adopts interest in the instantiated backwards-premises.

Reasoning backwards will build the *interest-graph* . The nodes of the *interest-graph* are either queries from *ultimate-epistemic-interests* or record sequents derivatively of interest. The latter will be referred to as "interests". The links either represent backwards reasons or connect sequents with queries for which they are answers. A link connects one node — the *resultant-interest* — with another interest in a sequent representing the first as yet unestablished backwards-premise of the backwards reason. Backwards reasoning begins with the resultant-interest and works backwards to the link-interest, adding it to the *interest-graph* if necessary. When the link-interest is established by drawing a conclusion, the conclusion is added to the list of supporting-nodes for the link. If there are no backwards-premises remaining to be established, the resultant-interest is inferred from the supporting-nodes. If there are backwards-premises remaining to be established, a new interest-link is constructed having the first of them as its link-interest. This is repeated until all the backwards-premises are established.

When an argument is found for a link-interest, the reasoner checks to see whether it is a deductive argument. A deductive argument is one in which none of the reasons employed are defeasible, no input states are used as premises, and no contingent premises are used. If the argument is deductive, the search for further arguments is cancelled. But if the argument is not deductive, then rather than cancelling the search for further arguments, the priority of the search is lowered.

To accommodate all of this, the *interest-links* will be a set of links *L* encoding the following information:

- the resultant-interest — the *interest-graph* node from which the backwards reasoning proceeds, or the query in *ultimate-epistemic-interests* for which the link records an answer.
- the link-interest — an interest in the first remaining unestablished backwards-premise.
- link-interest-formula — the interest-formula of the resultant-interest.
- link-interest-condition — an optional condition that a previously existing interest must satisfy to be the link-interest (rather than a new link-interest being constructed when the link is constructed).
- the link-rule — an indication of the rule of inference or substantive reason employed in such an inference, or ":answer" if the resultant node is a query. In the latter case, the link is an "answer-link".
- the remaining-premises — the list of backwards-premises (other than that represented by the link-interest) of the link-rule that are not yet established by nodes in the list of supporting-nodes.
- supporting-nodes — a list of inference-nodes validating the corresponding premises (backwards or forwards) and of adequate strength and satisfying any applicability-conditions for the premises.
- link-defeaters — if the link-rule is defeasible, this is the pair of c-lists for the rebutting-defeater and undercutting-defeater for the inference described by the link. When the link is created, these c-lists are created with empty lists of c-list-nodes if they do not already exist.
- the link-defeat-status — T if some inference-node in the c-list-nodes of one of the link-defeaters has a supposition contained in the resultant-supposition and an undefeated-degree-of-support greater than or equal to the degree-of-interest in the resultant-sequent; NIL otherwise. This represents a partial computation of defeat-status, proceeding just in terms of the validated members of the link-basis and the link-defeaters. It is used to lower the priority assigned to the backwards

reasoning if *L* is defeated (according to this partial computation).

- link-strength — the maximum-degree-of-interest conveyed by the link.
- link-generating-node — if the link is generated by a supposition (e.g., in conditionalization), this is the node encoding the supposition.
- discharged-link — T if the link has been discharged by finding an inference-node that validates the link-interest, NIL otherwise.

Interests will encode the sequents related by the *interest-link*s. This will not be taken to include the queries from *ultimate-epistemic-interests*. Thus the *interest-graph* actually includes both the members of *interests* and the queries from *ultimate-epistemic-interests*. An interest will be a data structure encoding the following information:

- the interest-sequent.
- the interest-formula — sequent-formula of the interest-sequent.
- the interest-supposition — sequent-supposition of the interest-sequent.
- the right-links — the list of *interest-links* in which the interest-sequent is a member of the link-basis.
- the left-links — the list of *interest-links* for which the present interest is the resultant-interest.
- the degree-of-interest in the interest-sequent — a number between 0 and 1; for non-terminal nodes this is the minimum of the degrees of interest of the interests to which the given interest is linked by right-links.
- the last-processed-degree-of-interest — if the reasoner has already attempted backwards reasoning from the interest-sequent with a lower degree-of-interest, this records that degree-of-interest.
- the interest-defeat-status — NIL if the defeat-status of any of the right-links is NIL, T otherwise.
- discharged-degree — the maximum degree of interest such that DISCHARGE-INTEREST-IN has been run on this interest with that maximal-degree-of-support. This is used in computing priorities.
- deductive-interest — T if the objective of the interest is to find a deductive proof of the interest-sequent; NIL otherwise.
- cancelled-interest — T if a deductive proof has been found for the interest-sequent; NIL otherwise.
- interest-queue-node — if the interest is currently on the *inference-queue*, this lists the queue-node enqueing it.
- interest-i-list — the i-list whose i-list-formula is the sequent-formula of the interest-sequent.
- maximum-degree-of-interest — for interests having right links, this is the minimum of the degrees of interest of the nodes to which the given interest is linked by right-links; for other interests this is the degree-of-interest of the node.
- interest-defeatees — if the node is of interest as a defeater for some *inference-graph* nodes, this lists those nodes.
- generated-suppositions — the list of inference-nodes having the interest as a generating-interest.
- generating-nodes — the list of inference-nodes encoding suppositions that give rise directly to the interest.
- interest-priority — the interest-priority used in ordering the *inference-queue*.
- discharge-condition — a condition that an inference-node must satisfy to discharge the interest.
- cancelling-node — the inference-node (if any) that discharges the interest resulting in its cancellation.
- interest-supposition-nodes — a list of inference-nodes
- generated-interest-schemes — the list of interest-schemes having this interest as their target-interest.
- generating-defeat-nodes — the list of inference-nodes for which this is an interest in a defeater.
- cancelled-left-links — the list of cancelled interest-links for which this is the

resultant-interest.

Some nodes of the *interest-graph* will be termini. They represent the starting points for backwards reasoning rather than being *for* something else. These will be either queries from *ultimate-epistemic-interests* or interests in defeaters for defeasible inference steps performed by the reasoner. Terminal nodes are distinguished by having empty lists of right-links.

As in the case of inference-nodes, interests will be stored in the list *interests*, which is a list of i-lists. An i-list is a data structure encoding the following information:

- i-list-formula.
- corresponding-c-list — the c-list, if any, whose c-list-formula is the same as the i-list-formula.
- i-list-interests — the list of interests in sequents having the i-list-formula as their sequent-formula.
- i-list-queries — the list of queries in *ultimate-epistemic-interests* having the i-list-formula as their query-formula.

Reasons

Forwards-reasons will be data-structures encoding the following information:

- reason-name.
- reason-function — explained below.
- forwards-premises — a list of forwards-premises.
- backwards-premises — a list of backwards-premises.
- backwards-premises-function — a function which, when applied to a binding of the reason-variables, produces the list of formulas determined by applying the binding to the backwards-premises.
- reason-conclusions — a list of formulas.
- conclusion-function — an optional function that computes the conclusions.
- reason-variables — the set of variables to be used in finding instances of the premises by pattern-matching.
- defeasible-rule — T if the reason is a defeasible reason, NIL otherwise.
- reason-strength — a real number between 0 and 1, or an expression containing some of the reason-variables and evaluating to a number.
- discount-factor — a real number between 0 and 1, used in prioritizing reasoning involving the use of the forwards-reason,
- reason-description — an optional string describing the reason.

The discount-factor will usually be 1, but by setting it lower we can lower the priority of reasoning using this reason. This will be illustrated in chapter four.

Forwards-premises are data-structures encoding the following information:

- fp-formula — a formula.
- fp-kind — :inference, :percept, or :desire (the default is :inference)
- fp-condition — an optional constraint that must be satisfied by an inference-node for it to instantiate this premise.
- fp-variables — the reason-variables occurring in this premise.
- fp-binding-function — a two-valued function which, when applied to a formula that instantiates the premise, returns the list of ordered pairs (an a-list) telling how the fp-variables are bound to obtain this instance, and also returns the instantiation of the conclusions-variables (see chapter five).
- fp-instantiator — a function which, when applied to a premise and a binding returns the formula produced by substituting the bindings for the variables in the premise.

Similarly, *backwards-premises* are data-structures encoding the following information:

- bp-formula
- bp-kind
- bp-condition1 — an optional predicate that formulates a constraint that must be

satisfied by an existing interest to which the backwards-reason is being.
- bp-condition2 — an optional predicate which should be such as to set the values of slots so that the resulting interest satisfies bp-condition1.
- bp-instantiator

The use of the premise-kind is to check whether the sequent from which a forwards inference proceeds represents a desire, percept, or the result of an inference. The contents of percepts, desires, and inferences are all encoded as sequents, but the inferences that can be made from them depend upon which kind of item they are. For example, we reason quite differently from the desire that *x* be red, the percept of *x*'s being red, and the conclusion that *x* is red.

Forwards-reasons are most easily defined using the macro DEF-FORWARDS-REASON, which has the following form:

```
(def-forwards-reason symbol
   :forwards-premises list of pretty-formulas optionally interspersed with with expressions of the form
      (:kind ...) or (:condition ...)
   :backwards-premises  list of pretty-formulas optionally interspersed with with expressions of the form
      (:kind ...) or (:condition1 ...) or (:condition2 ...) or (:condition ...)
   :conclusions  list of formulas
   :conclusions-function lambda expression or function
   :strength number or a lambda-expression evaluating to a number whose single variable is 'binding'
      (default is 1.0).
   :variables list of symbols
   :defeasible? T or NIL (NIL is default)
   :description an optional string (quoted) describing the reason)
```

For example:

```
(def-forwards-reason *DISCOUNTED-PERCEPTION*
   :forwards-premises
   "((the probability of p given ((I have a percept with content p) & R)) <= s)"
      (:condition (< s 0.9))
   "(p at time)"
      (:kind :percept)
   "(R at time0)"
      (:condition (projectible R))
   :backwards-premises
   "(time0 < time)"
   :conclusions "(p at time)"
   :variables p time R time0 s
   :strength #'(lambda (binding) (cdr (assoc 's binding)))
   :defeasible? t
   :description "When information is input, it is defeasibly reasonable to believe it.")
```

In defining reasons, *:condition1* and *:condition2* are most often used to define a discharge-condition. That can be done more simply by using *:condition*. Conditions frequently involve arithmetical relations between the variables. In formulating conditions, '+', '-', '*', '/', '<', '<=', and '=' can be written in either infix form or prefix form.

Reasons are applied by pattern-matching, using the reason-variables as the variables for the pattern-match. For example, modus-ponens can be defined as follows:

```
(def-forwards-reason modus-ponens
     :forwards-premises  "P"  "(P -> Q)"
     :conclusions  "Q"
     :variables  P Q)
```

Then given the conclusion (A & B) and ((A & B) -> (C v D)), the reason-variable P can be bound to "(A & B)" and Q to "(C v D)", with the result that these formulas match the forwards-premises. Then the binding is applied to the conclusions to yield "(C v D)", which is then inferred. In chapter five, when free variables and unification are introduced, this will become more complicated, but for now we can use simple pattern-matching. This is done efficiently by constructing a function that compiles the pattern-match at the time the reason is defined. Sometimes efficiency can be increased by hand-coding that function and storing it in the slot *reason-function*.

Normally, the conclusions of a forwards-inference are constructed by applying the pattern-match produced by the premises to the conclusions, but sometimes the conclusions should be some more complex function of that pattern match. The conclusions-function provides the machinery for this. For instance, we might define a somewhat whimsical reason as follows:

```
(def-forwards-reason GREEN-SLYME
   :forwards-premises
     "(Patient-has-had-green-slyme-running-out-of-his-nose-for  x  hours)"
     (:condition (numberp x)
      (:kind inference))
   :variables   x
   :conclusions-function
     (The-probability-that the-patient-ingested-silly-putty is (* .5 (- 1 (exp (- x)))))
   :strength   .95
   :defeasible? T)
```

Backwards-reasons will be data-structures encoding the following information:

- reason-name.
- reason-function.
- forwards-premises.
- backwards-premises.
- reason-conclusions — a list of formulas.
- reason-variables — the set of variables to be used in finding instances of the premises by pattern-matching.
- strength — a real number between 0 and 1, or or an expression containing some of the reason-variables and evaluating to a number.
- defeasible-rule — T if the reason is a defeasible reason, NIL otherwise.
- reason-condition — a condition that must be satisfied bythe sequent of interest before the reason is to be deployed.
- reason-discharge — for rules like *conditionalization* that introduce new suppositions, this is the formula supposed.
- reason-length — the number of backwards-premises.
- discount-factor — a real number between 0 and 1, used in prioritizing reasoning involving the use of the forwards-reason.
- conclusion-variables — the variables occurring in the reason-conclusions.
- conclusion-binding-function — a function which, when applied to an interest having the form of the first member of the reason-conclusions, produces the binding (an a-list) of the conclusion-variables.

Backwards-reasons are most easily defined using the macro DEF-BACKWARDS-REASON, which has the following form:

```
(def-backwards-reason symbol
   :conclusions  list of formulas
   :forwards-premises list of pretty-formulas optionally interspersed with with expressions of the form
      (:kind ...) or (:condition ...)
   :backwards-premises  list of pretty-formulas optionally interspersed with with expressions of the form
      (:kind ...) or (:condition1 ...) or (:condition2 ...) or (:condition ...)
   :conclusions-function lambda expression or function
   :condition  this is a predicate applied to the binding produced by thetarget sequent
   :strength number or a lambda-expression evaluating to a number whose single variable is 'binding'
     (default is 1.0).
   :variables list of symbols
   :defeasible? T or NIL (NIL is default)
   :description an optional string (quoted) describing the reason)
```

For example:

```
(def-backwards-reason *CAUSAL-IMPLICATION*
   :conclusions  "(Q at time*)"
   :forwards-premises
   "(A when P is causally sufficient for Q after an interval interval)"
      (:condition (every #'temporally-projectible (conjuncts Q)))
   "(A at time)"
   :backwards-premises
   "(P at time)"
   "((time + interval) < time*)"
   :variables  A P Q interval time time*
   :defeasible?  T)
```

The macros def-forwards-undercutter and def-backwards-undercutter facilitate the definition of undercutting defeaters. These work just like def-forwards-reason and def-backwards-reason, except that rather than listing the conclusions, we list the defeated reason. For example:

```
(def-backwards-undercutter *DEFEAT-FOR-CAUSAL-IMPLICATION*
   :defeatee *causal-implication*
   :backwards-premises
   "((Q at time1) @ (Q at time*))"
  "((time + interval) < time1)"
   "(time1 < time*)"
   :variables time time1 time* interval A P Q)
```

The inference-queue

The *inference-queue* will enqueue inference-nodes encoding input states (which will be percepts and primitive desires), new inference-nodes encoding the results of inferences, queries from *ultimate-epistemic-interests,* and new interests (*interest-graph* nodes). Accordingly, I will take the *inference-queue* to be an ordered list of data-structures called *inference-queue-nodes*, each encoding the following information:

- the item enqued — an inference-node, a query, or an interest;
- the item-kind — :conclusion, :interest, or :query;
- item-complexity — discussed below;
- discounted-strength —discussed below;
- the degree-of-preference — discussed below.

The ordering of the nodes of the *inference-queue* will be done by the relation ⌜*x is i-preferred to y*⌝. Specifying this relation is part of the process of programming OSCAR. A simple preference relation that is used as the default appeals to two properties of the enqued item. The first will be referred to as the "discounted-strength" of the item, and will be measured by a real number between 0 and 1. The discounted-strength of a desire is simply its desire-strength. The discounted-strength of a percept is its clarity. The clarity of a percept will, for now, be treated as a primitive property whose value is specified by the perceptual input systems to which OSCAR is attached. This may be treated in a more sophisticated way later. The discounted-strength of an inference-node of type "inference" will, in most cases, be measured by the undefeated-degree-of-support of the inference-node, but there are two complications. First, inference-nodes recording suppositions are generated by interests. For example, interest in a conditional will lead the reasoner to suppose the antecedent with the objective of inferring the consequent, and the supposition of the antecedent is recorded as an *inference-graph* node and an inference-node. The degree-of-jusitification of this inference-node will automatically be 1, but the priority assigned to reasoning from it should be a function of the discounted-strength of the generating interest. Accordingly, when an inference-node is generated by an interest, its discounted-strength will be taken to be the discounted-strength of the interest rather than the undefeated-degree-of-support. Second, it turns out to be desirable to give inferences in accordance with some inference rules (either backwards or forwards) low priority. This is accomplished by attaching a *discount-factor* (a real number between 0 and 1) to the rule, and using that to lower the degree-of-preference. Accordingly, the discounted-strength of an inference-node of type "inference" that is not generated by an interest will, in general, be the maximum (over its node-arguments) of the product of the discount-factor of the support-link-rule of the last support-link in the argument and the strength of the argument. (This is computed recursively by COMPUTE-UNDEFEATED-DEGREES-OF-SUPPORT, without computing the list of node-arguments.) The discounted-strength of an interest will be its interest-priority, which will be discussed below.

The second property to which the default preference relation appeals is "complexity". This is a measure of syntactic complexity, and the idea behind the preference ordering is that reasoning with simpler formulas is given precedence over reasoning with complex formulas. Accordingly, we define:

> Where *x* is an enqued item, *strength* is the discounted-strength of *x*, and *complexity* is the complexity of the sequent expressing the content of *x*, *the degree-of-preference of x* is *strength/complexity*.

The default preference relation is then:

> *x is i-preferred to y* iff the degree-of-preference of *x* is greater than the degree-of-preference of *y*.

With one qualification, the default complexity measure simply counts the number of symbol tokens (not counting parentheses) occuring in the sequent expressing the content of the enqued item. This has the effect of giving precedence to reasoning with short formulas. The qualification is that non-reductio-suppositions (explained in chapter four) are not counted in computing complexity. (In first-order reasoning, they will only be ignored if they contain no free variables or skolem-functions.) The only justification I have to offer for this complexity measure is that it seems to work reasonably well, and it is a rough reflection of the way human beings prioritize their reasoning.

The discounted-strength of an interest is its interest-priority. To understand this concept, it must first be realized that degrees of interest play two roles. They determine how justified an answer must be for it to be an acceptable answer, and they prioritize reasoning. Initially, the priority of backwards reasoning is determined directly by the degree-of-interest. If something is of interest ultimately for answering a certain query, and an answer is obtained for that query, this lowers the priority of backwards reasoning from that interest, but does not lower the degree-of-interest itself (in its role of determining how good answers must be). So the *interest-priority* of an interest should be some function of (1) the degree-of-interest, (2) the discount-factor of the backwards reason used in its derivation, (3) whether the interest is ultimately aimed at answering a query none of whose query-answers are believed, (4) whether the interest derives from a defeated member of the *interest-graph*, and (5) whether the node-sequent is already supported by an inference-node whose undefeated-degree-of-support is greater than or equal to the degree-of-interest. The interest-priority must also be a number between 0 and 1. Supplying the exact definition of interest-priority will be part of the process of programming OSCAR, but a default definition is built into OSCAR. This definition is the following:

- The interest-priority of a query from *permanent-ultimate-epistemic-interests* is its degree-of-interest.

- The interest-priority of any other query is α_0 if it has already been answered, and its degree-of-interest otherwise.
- The interest-priority of an *interest-graph* node having an empty set of right-links is α_0 (this represents interest in a defeater for a defeasible inference);
- The interest-priority of any other *interest-graph* node *node* is computed as follows. Let *L* be the set of its undefeated right-links having resultant-interests with maximum-degrees-of-interest greater than the maximum undefeated-degree-of-support at which they have been discharged. If *L* is empty, the interest-priority of *node* is α_0; otherwise it is the maximum over *L* of the set of products of the discount-factor of the link-rule and the interest-priority of the resultant-interest.

α_0 serves as a base-level of interest-priority used for reasoning from defeated nodes and used for the search for defeaters for *inference-graph* nodes. As long as $\alpha_0 > 0$, the reasoner will “eventually” get around to reasoning prioritized in this way, provided new environmental input does not continually delay it by inserting new items in front of it in the *inference-queue*. The higher the value of α_0, the more effort the reasoner is apt to expend on these matters.

4. Epistemic Cognition

Epistemic cognition is performed by THINK:

THINK

If the *inference-queue* is nonempty, let *Q* be the first member of the *inference-queue*:

- Remove *Q* from the *inference-queue*.
- Let *node* be the item enqued by *Q* if it is an inference-node:
  - insert *node* into the list of *processed-conclusions*.
  - REASON-FORWARDS-FROM *node*
- Let *query* be the item enqued by *Q* if it is a query, and let *priority* be its interest-priority:
  - REASON-BACKWARDS-FROM-QUERY *query priority*.
- Let *interest* be the item enqued by *Q* if it is an *interest-graph* node, and let *priority* be its interest-priority:
  - REASON-BACKWARDS-FROM *interest priority*.
  - FORM-EPISTEMIC-DESIRES-FOR *interest*.

If the list *new-links* is nonempty, UPDATE-BELIEFS and set *new-links* to $\varnothing$.

Desires are enqued as *inference-graph* nodes, and as such are used for forwards reasoning rather than backwards reasoning. This might seem surprising, but recall that desires function as forwards directed prima facie reasons for concluding that the desired state is a suitable goal.

4.1 Reasoning Backwards

Reasoning backwards begins by reasoning backwards from queries contained in *ultimate-epistemic-interests*. Reasoning backwards from a query consists of constructing an interest to record interest in the query-formula and then reasoning backwards from it:

REASON-BACKWARDS-FROM-QUERY *query priority*

Construct an *interest* *N* encoding interest in the query-formula, with a right-link *L* to *query* with the degree-of-interest *degree* equal to the degree-of-interest in *query*. Then:

- REASON-BACKWARDS-FROM *N priority.*
- FORM-EPISTEMIC-DESIRES-FOR *N*.
- If *N* is validated to undefeated-degree-of-support *degree* by some inference-node *C*, place *C* in the list of supporting-nodes for *L*, and DISCHARGE-LINK *L degree degree*.

Backwards reasoning builds the *interest-graph*. It begins with interests recording interest in defeaters and recording queries from *ultimate-epistemic-interests*. Then it applies backwards reasons to generate interest-links. The left terminus of an interest-link will be a single interest which answers the query or a list of interests from which the resultant interest can be inferred (together with the forwards-premises) in accordance with some backwards-reason. Then the process will be repeated, generating new interest-links and new interests.

The structure of backwards reasoning is diagrammed in figure 14. Beginning with interest in a sequent *S*, backwards reasoning will first find all backwards reasons that would allow the inference of *S* with a maximal-degree-of-support greater than or equal to the degree-of-interest. It uses these to build interest-schemes, which result from partially instantiating the premises of the reason so that the conclusion is identical with the interest-formula of the target-interest. Interest-schemes are recorded in the list *interest-schemes*. For each such interest-scheme, the reasoner searches for all combinations of inference-nodes that comprise instances of the forwards-basis (the partially instantiated forwards premises), and uses them to generate fully instantiated instances of the backwards reason. This is the process of "discharging an interest-scheme". The resulting instances are recorded as interest-links.

The resultant-interest from which the backwards reasoning is proceeding could itself be the left terminus of several different interest-links. In that case, the degree-of-interest for the node is the minimum of the degrees-of-interest in the resultant-interests of those links. This ensures that backwards reasons of use for inferring *any* of those resultant-interests will be considered. On the other hand, the interest-priority will be the maximum of the interest-priorities of the resultant-interests. That way, the reasoner assigns appropriate interest to answering important questions, and if it answers less important ones along the way, so much the better. When a new left-link is constructed, the reasoner adopts interest in the first member of the remaining-premises. If an inference-node is found that validates the interest to an adequate undefeated-degree-of-support, it is "discharged" and the reasoner goes on to the next remaining-premise, constructing a new interest-link having that as the link-interest. Taking premises one at a time prevents the reasoner from expending its resources looking for arguments for the later premises if no argument can be found for the earlier ones. Once all of the premises are validated to an adequate undefeated-degree-of-support, then if the resultant-interest is an interest, its node-sequent will be inferred from the supporting-nodes of the link. This is handled by MAKE-BACKWARDS-INFERENCE. If instead the resultant-interest is a query, DISCHARGE-ULTIMATE-INTEREST will be applied to it.

Two things can happen when the reasoner adopts interest in the next remaining-premise. First, there might already be an interest recording interest in that sequent. In that case, the left-link is simply attached to it as a new right-link, and the degree-of-interest is recomputed. If the interest is still on the *inference-queue*, its interest-priority must also be recomputed. If the interest is no longer on the *inference-queue*, and the new link lowers its degree-of-interest, then it must be reinserted into the *inference-queue* and backwards reasoning from it reinitiated for those backwards reasons that were too weak to satisfy the original interest. Second, there might be no pre-existing interest recording interest in the sequent. In that case, we must consider two possibilities. (1) In most cases, the reasoner will not yet have found an argument supporting the sequent of interest. In this case, a new interest is constructed recording the interest, and it is inserted into the *inference-queue*. But (2), it could happen that the reasoner has already found an argument validating the sequent of interest (and recorded it in the *inference-graph*). If the argument is deductive, there is no point in proceeding with this interest. The reasoner should go on to the next remaining-premise and adopt interest in it. But if the argument is not deductive, a new interest should be constructed just as if no argument had yet been found. The new inference-node should be placed on the *inference-queue*, but with low priority. That way, if the argument validating the sequent is subsequently defeated, the priority will be recomputed and backwards reasoning from it will occur in the normal way. As there is already an argument validating the sequent of interest, the reasoner will then go on to the next remaining-premise and adopt interest in it, and so on.

REASON-BACKWARDS-FROM *interest priority*

- Let *sequent* be the interest-sequent of *interest*, δ its degree-of-interest, and δ* its last-processed-degree-of-interest.
- For each backwards-reason *R* of strength at least δ but less than δ*, if *interest* is still uncancelled and the (first) conclusion of *R* matches the interest-formula of *interest* by pattern-matching using the reason-variables, and *binding* is the resulting assignment of values to the reason-variables, apply MAKE-BACKWARDS-INFERENCE to *R*, *binding*, and *interest*.
- For each auxiliary rule for backwards reasoning *R*, apply *R* to *S*.

MAKE-BACKWARDS-INFERENCE *R binding interest*

- if *R* has no forwards-premises, construct an interest-link for inferring *sequent* from the backwards-premises, instantiating the premises by applying *binding* to them, and apply DISCHARGE-LINK to *link* δ;
- if *R* has forwards-premises, then construct an interest-scheme for inferring *sequent* from the forwards- and backwards-premises collectively, partially instantiating the premises by applying *binding* to the interst-formulas and for each complete instantiation of the forwards-premises such that all are validated by processed-conclusions with maximal-degree-of-support at least δ, and *max-degree* is the maximum-degree-of-interest of *interest*, make an interest link *link* recording the instance of the interest-scheme. Let the discharge-condition of the link-interest be the same as the fp-condition2 for the premise, and apply DISCHARGE-LINK *link* δ.

Discharging a link is done first when the link is constructed and consists of looking for inference-nodes of adequate strength validating members of the link-interest. If such a node is found and the link has remaining-premises, a new interest-link is constructed by inserting the validating node into the list of supporting-nodes of the discharged link, deleting the first remaining-premise from the list of remaining-premises and making it the link-interest. Then the new link is discharged. If instead the link has no remaining-premises, then (1) if the link is an answer-link, the new inference-node is recorded as an answer to the query that is the resultant-interest of the link, and (2) otherwise, the resultant-sequent is inferred from the supporting-nodes of the link.

DISCHARGE-LINK *link* δ (using *priority*, if available, to simplify the computation of interest-priorities):

- For every inference-node *N* in the c-list-nodes of the correspondingc-list of the interest-i-list of the resultant-interest of *link*, if *N* satisfies the discharge-condition (if any) of the resultant-interest and the node-supposition of *N* is a subset of the interest-supposition of the resultant-interest
    - If *link* is an answer-link, add *N* to the list of supporting-nodes of *link*□ and add *N* to the list of query-answers for the resultant-interest of *link*.
    - If *link* is not an answer-link and has no remaining-premises, where *discharge* is the link-discharge of *link* and *R* is the link-rule, let *formula* be the interest-formula of the resultant-interest of *link*, and DRAW-CONCLUSION *formula supporting-nodes R discharge*.
    - If instead *link* has remaining-premises, construct a new interest-link by inserting the validating node into the list of supporting-nodes of the discharged link, deleting the first remaining-premise from the list of remaining-premises and making it the link-interest. Apply DISCHARGE-LINK to the new-link and δ.
    - If *N* is a deductive-node, CANCEL-INTEREST-IN *interest*.
    - If *N* is not a deductive-node, then for all generated-suppositons of *interest* that have no other generating-interests, if *N* is deductive-in the supposition, apply CANCEL-NODE to the supposition.

Cancellation will be discussed below.

Some backwards-reasons will instruct the reasoner to make a new supposition. For example, *conditionalization* instructs the reasoner that if it is interested in a conditional $(P \rightarrow Q)$, it is to suppose P and try to infer $Q/\{P\}$. Supposing P consists of constructing an inference-graph node for the sequent $P/\{P\}$, storing it in *conclusions,* and inserting it into the *inference-queue*.

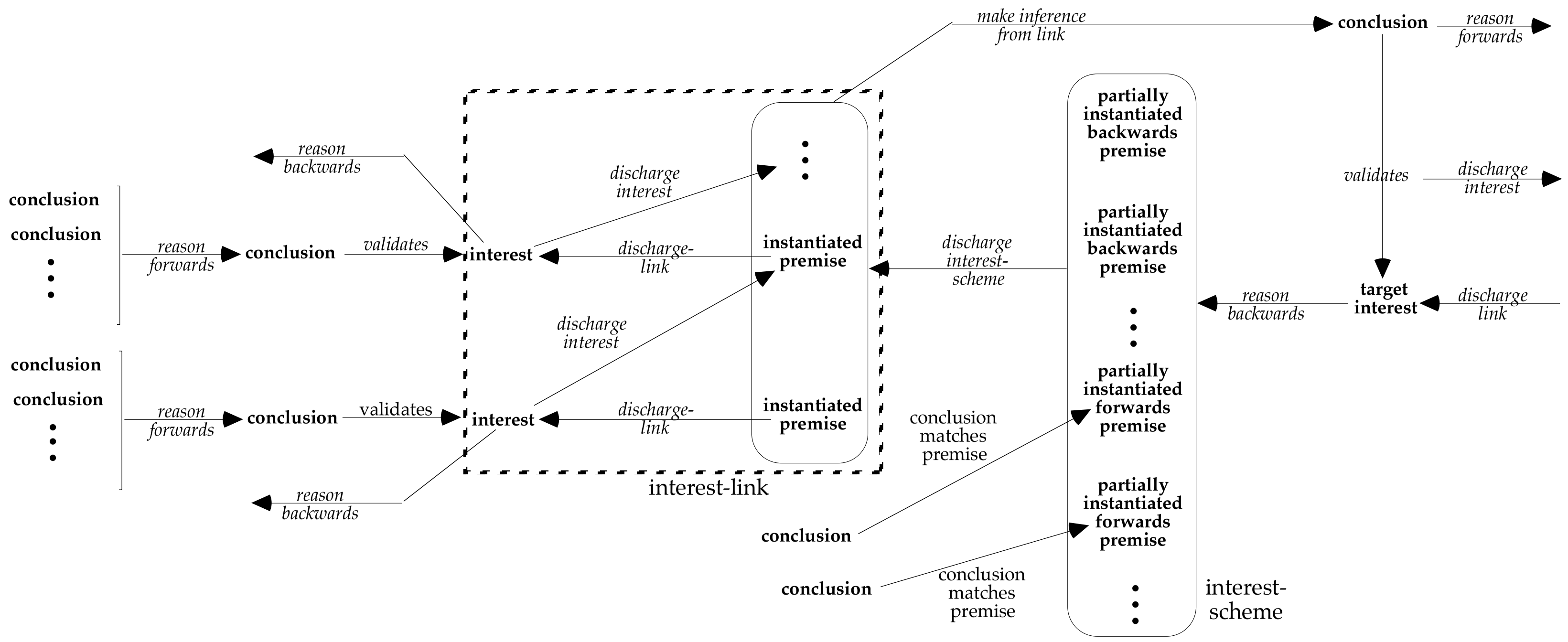

Figure 14. Backwards reasoning and the discharge of interest-links.

4.2 Reasoning Forwards

Reasoning forwards consists of three operations. First, new inference-nodes are used to instantiate forwards-reasons, and inferences made on that basis. Second, new inference-nodes are used to instantiate the forwards-premises of interest-schemes, thereby producing new interest and interest-links for use in backwards reasoning. These first two kinds of inference are performed when the inference-node is retrieved from the *inference-queue*. Third, new inference-nodes are used to discharge interests that they validate. This operation is performed immediately upon producing the new inference-node rather than waiting until it is retrieved from the *inference-queue*. This allows one interest-discharge to lead immediately to another, and thereby allows the reasoner to rapidly discharge an entire chain of backwards reasoning and get back to the queries that intiated it, without having to perform a lot of unrelated reasoning that might be interspersed on the *inference-queue*.

The forwards reasoning that is performed when an inference-node is retrieved from the *inference-queue* is performed by REASON-FORWARDS-FROM. The forwards reasoning that is performed immediately upon producing a new inference-node is performed by DRAW-CONCLUSION.

REASON-FORWARDS-FROM *node*

- DISCHARGE-INTEREST-SCHEMES *node* .
- APPLY-FORWARDS-REASONS *node*.
- If (interests-discharged? *node*) is NIL, apply DISCHARGE-INTEREST-IN to *node* and the corresponding-i-list of the node-c-list of *node*.
- For each auxiliary rule *R* for forwards reasoning, apply *R* to *node*.

Auxiliary rules are included for the sake of generality. These are rules that are triggered by adopting new inference-nodes but do not simply make inferences from the new inference-nodes. We also allow auxiliary rules for backwards reasoning.

APPLY-FORWARDS-REASONS *node*

For each forwards-reason *R*, if there is a list *nodes* such that (1) *node* is in *nodes*, (2) the other members of *nodes* are processed conclusions, (3) the members of *nodes* satisfy the applicability-conditions of *R*, and (4) where *B* is the list of node-formulas of the members of *nodes*, *B* matches the forwards-premises of *R* by pattern-matching, and *binding* is the resulting assignment to the reason-variables, then MAKE-FORWARDS-INFERENCE *R binding basis*.

MAKE-FORWARDS-INFERENCE *R binding basis*

- If *R* has no backwards-premises and *P* is the result of applying *binding* to one of the conclusions of *R*, DRAW-CONCLUSION *P nodes R*.
- If *R* has backwards-premises, adopt interest in the result applying *binding* to the conclusions of *R* (but do not reason backwards from the new interests), and for each such new interest construct a left-link from it to the result of applying *binding* to the backwards-premises of *R* and apply DISCHARGE-LINK to that link.

DRAW-CONCLUSION is the operation that records the result of a forwards inference in new *inference-graph* nodes, and performs whatever subsidiary operations we want to immediately apply to these nodes. These operations consist of (1) constructing the inference-node supporting the node-sequent and the c-list for the node-formula, if these do not already exist, (2) if the inference-node is newly-constructed, determining and recording what queries, if any, are answered by it, (3) deleting logically-weaker subsumed nodes (see below) from the databases used by the reasoner, (4) recomputing defeat-statuses and *inference-queue* priorities, and (5) if the inference-node is either newly-constructed or its maximal-degree-of-support increases as a result of adding the new nodes to its node-list, it is used to discharge interests that it is now but was not previously strong enough to satisfy.

DRAW-CONCLUSION *P B R discharge (optional-argument: interest)*

If no member of *B* is cancelled, let *sup* be the union of the node-suppositions of the members of *B*, less *discharge*, and let *S* be the sequent ⟨*sup,P*⟩. If *S* has not been validated deductively:

- If *R* is not defeasible, let *NDA* be the set of unions of the crossproducts of the sets of nearest-defeasible-ancestors of members of *B**, and otherwise let *NDA* be NIL.
- If either *R* is defeasible or not SUBSUMED *S NDA* :
  - If there is an inference-node of kind :inference supporting *S*, let *node* be that inference-node.
  - Otherwise, construct a new inference-node *node* supporting *S*.
  - If *node* is a deductive-node, and *interest* is not NIL, add *interest* to the list of enabling-interests for *node*.
  - Build a support-link *link* recording the inference of *node* from *B* in accordance with *R*, and recompute the lists of node-ancestors and nearest-defeasible-ancestors for the inference-descendants.
  - If the preceding step does produce new arguments for *node*:
    - ADOPT-INTEREST-IN-DEFEATERS-FOR *link*.
    - Record *link* as a support-link for *node*.
    - If *node* is newly constructed, add it to the *inference-graph* and store it in the list of *conclusions*
    - If *R* is defeasible, add {*node*} to the list of nearest-defeasible-ancestors of *node*; otherwise append *NDA* to the list of nearest-defeasible-ancestors of *node*.
    - CANCEL-SUBSUMED-LINKS *link*.
    - If *node* is either newly-constructed, or its maximal-degree-of-support has increased as a result of adding this support-link, let *old-degree* be the old maximal-degree of support for *node*:
      - Let *i-list* be the corresponding i-list of the conclusion-c-list of *node*.
        - If *node* is either newly-constructed, or its maximal-degree-of-support has increased as a result of adding this support-link, DISCHARGE-INTEREST-IN-DEFEATERS *node i-list old-degree*
        - Add *ink* to the list *new-links*.
        - If *node* already existed but this inference increases the maximal-degree-of-support of *node*, ADJUST-SUPPORT-FOR-CONSEQUENCES *node old-degree*
        - If *node* is either newly-constructed, or its maximal-degree-of-support has increased as a result of adding this support-link, DISCHARGE-INTEREST-IN *node i-list old-degree interest*

Let us say that one support-link L_1 *subsumes* a secondL_2 when they are so-related that (1) no conclusion can be inferred from the support-link-target of L_2 that cannot be inferred from the support-link-target of L_1 and (2) anything defeating L_1 would also defeat L_2 . In such a case, L_2 is redundant and that inference should not be made. If it has already been made when L_1 is constructed, then it should be cancelled. To encode these subsumption rules, we need a precise characterization of (sufficient) conditions for subsumption. OSCAR employs three such conditions:

(a) If L_1 and L_2 are non-defeasible support-links for the same node, then L_1 subsumes L_2 if every set *X* in the list of nearest-defeasible-link-ancestors for L_2 contains as a subset some set *Y* in the list of nearest-defeasible-link-ancestors for L_1.

(b) Let N_1 be the support-link-target of L_1 and N_2 the support-link-target for L_2. If N_1 and N_2 have the same node-formula, and the node-supposition of N_1 is a subset of the node-supposition of N_2, and either

(i) L_2 records a defeasible inference, L_1 records an infrence in accordance with the same inference-rule, the union of the nearest-defeasible-ancestors of the basis of L_1 is the same as the union of the nearest-defeasible-ancestors of the basis of L_2 (so that they will be defeated by the same defeaters), and the the binding employed in inferring L_2 equals the binding employed in L_1 (so the defeaters are the same except possibly for L_2 having defeaters containing additional formulas in their suppositions); or

(ii) L_2 records a non-defeasible inference and every set X in the list of nearest-defeasible-link-ancestors for L_2 contains as a subset some set Y in the list of nearest-defeasible-link-ancestors for L_1,

then L_1 subsumes L_2.

Blocking or removing subsumed support-links is accomplished by the "not SUBSUMED" restriction in DRAW-CONCLUSION, which precludes creating new support-links violating this restriction, and the operation CANCEL-SUBSUMED-LINKS, which eliminates pre-existing support-links violating lthis restriction.

In searching for defeaters for a defeasible inference, we want to (1) be alert for defeaters for any argument (regardless of strength) in which the inference is used, and (2) minimize the priority of the search. This can be accomplished by taking the degree-of-interest in defeaters to be 0 and the interest-priority to be α_0:

ADOPT-INTEREST-IN-DEFEATERS-FOR *S B R new-nodes*

- If *R* is a defeasible reason, insert an interest in the rebutting defeater and the undercutting defeater for the inference from *B* to *S* into the *inference-queue*, taking the degree-of-interest to be 0 and the interest-priority to be α_0.
- Find any existing defeaters for the members of *new-nodes*, and record them in the lists of node-defeaters and node-defeatees for the respective nodes.
- If *S* is inferred via a Q&I module, adopt interest in defeaters for that inference.

When a new support-link is added to an existing node, that can increase its maximal-degree-of-support, and those for it node-consequences, and the node-consequences of the node-consequences, and so on recursively. These changes are computed by ADJUST-SUPPORT-FOR-CONSEQUENCES. Furthermore, each time this increases the maximal-degree-of-support for a node, DISCHARGE-INTEREST-IN-DEFEATERS, and DISCHARGE-INTEREST-IN, must be run for the new maximal-degree-of-support to catch any discharges that could not be performed for the previous maximal-degree-of-support but can now be performed. If the node is a processed-conclusion, DISCHARGE-INTEREST-SCHEMES must also be run. (This difference reflects the fact that DISCHARGE-INTEREST-IN is applied to inference-nodes immediately upon creation, while DISCHARGE-INTEREST-SCHEMES is only applied to inference-nodes when they are retrieved from the *inference-queue*).

Cancellation rules are very important in OSCAR. These prevent OSCAR from performing reasoning that is no longer of use. DISCHARGE-INTEREST-IN and DISCHARGE-LINK both apply two cancellation rules. First, if an interest is established deductively, then there is no reason to seek further arguments for it, so it is removed from the list of interests. Second, let us say that one node is *deductive-in* a second node if the first node is a node-ancestor of the second, and the first node and every inference-ancestor of it having the second node as an inference-ancestor is obtained by making a deductive inference. In other words, the first node is deductive-in the second iff the second is used in reasoning to the first, and all such reasoning is deductive. If an interest generates a supposition, and the interest is subsequently obtained by an argument that is deductive-in that supposition, then that supposition has served its purpose and there is no reason to pursue further reasoning from it. This is accomplished by cancelling the node recording the supposition. (The details of cancelling a node will be discussed below).

In the following *new?* is T iff *node* is newly-constructed, and if *node* is not newly-constructed then *old-degree* is its previous maximal-degree-of-support, and *i-list* is the corresponding-i-list of the node-c-list for *node*:

DISCHARGE-INTEREST-IN *node i-list old-degree new? (optional-argument: interest)*

If either *new?* is T or the maximal-degree-of-support of *node* is greater than *old-degree*:

- Let δ be the maximal-degree-of-support of *node*.
- For each interest *N* in the list of i-list-interests of *i-list*, if *N* is *interest*, or (1) either *new?* is T, or δ is greater than *old-degree*, (2) the degree-of-interest of *N* is less than

or equal to δ, (3) *node* satisfies the discharge-condition of *N*, and (4) *node* validates the node-sequent of *N*, then:
- For all right-links *link* of *N*, if the degree of interest in the resultant-interest of *link* is less than or equal to δ, and either *new*? is T or the link-strength of *link* is greater than *old-degree*:
  - If the link-rule of link is "answer" (i.e., it is a link to a query), add *node* to the list of supporting-nodes of *link* and DISCHARGE-LINK *link*.
  - Otherwise, DISCHARGE-LINK *link*.
- If *node* was established deductively, CANCEL-INTEREST-IN *N node*.
- Otherwise, if the node-supposition of *conclusion* is a subset of the interest-supposition of *interest*, then for any generated-suppositions *sup* of *interest* which are such that *node* is deductive in *sup*, CANCEL-NODE *sup node*.

This pseudocode implements an important observation. When the reasoner makes an inference from a link, the resulting conclusion is guaranteed to discharge the resultant-interest of the link. In that case, the resultant-interest is passed to DRAW-CONCLUSION as a new argument, and that argument is in turn passed to DISCHARGE-INTEREST-IN. The latter can then assume that the conclusion is appropriately related to that particular interest to discharge it.

DISCHARGE-INTEREST-IN-DEFEATERS *node i-list old-degree*
- Let δ be the maximal-degree-of-support of *node*.
- For each interest *N* in the list of i-list-interests of *i-list*, if the degree-of-interest of *N* is less than or equal to δ, the maximal-degree-of-interest for *N* is greater than *old-degree*, and *node* validates the node-sequent of *N*, then:
  - For each interest-defeatee *L* of *N*, add *node* to the list of node-defeaters of *L* and add *L* to the list of node-defeatees of *node.*

CANCEL-INTEREST-IN *N*
- Remove *N* from its i-list, and if this empties the list of i-list-interests, remove the i-list from the list of *interests*.
- Set the value of the slot *cancelled-interest* for *N* to T.
- If *node* records a generated-suppositions of *N*, CANCEL-NODE *node*.
- If *N* is on the *inference-queue,* delete it from the *inference-queue*.
- For each interest-scheme *S* whose instance-consequent is the node-sequent of *N*, delete *S* from *interest-schemes*.
- For all left-links *L* for *N*, delete *L* from the *interest-graph*, and for each link-interest *M* of *L*, remove *L* from the list of right-links of *M*.
- For each of those link-interests *M*, if no right-links remain, CANCEL-INTEREST-IN *M*.

CANCEL-NODE *node*
- Set the value of the slot *cancelled-node* for *node* to T.
- If *N* is a generated-interest of *node* that are not also generated in some other way ("unanchored interests"), CANCEL-INTEREST-IN *N* .
- If *N* is a generated-interest of *node*, remove *node* from the list of generating-nodes of *N*.
- If *N* is a generating-interest of *node*, remove *node* from the list of generated-suppositions of *N*.
- Remove *node* from the list of c-list-nodes of its c-list, and if this empties the list of c-list-nodes, remove the c-list from *conclusions* and set the corresponding-c-list for the corresponding-i-list to NIL.
- Remove *node* from the list of c-list-processed-nodes of its node-c-list.
- If *node** is a node-consequence of *node*, CANCEL-NODE *node** *protected-node*.
- If *node* is still on the *inference-queue,* remove it.

DISCHARGE-INTEREST-SCHEMES *node old-degree*

- Let *S* be the node-sequent of *node* and let δ be its maximal-degree-of-support.
- For each interest-scheme, if the degree of interest of the target-interest is ≥ *old-degree* and *S* is an instance of one of the remaining-premises, and that instantiation can be extended to an instantiation of all the remaining-premises in such a way that they are all validated with maximal-degree-of-support greater than *old-degree*, construct an interest-link for inferring the interest-sequent of the target-interest from the backwards-premises, instantiating the premises in accordance with the instantiation of the forwards-premises, and apply DISCHARGE-LINK to *link*.

Recall that UPDATE-BELIEFS is as follows:

UPDATE-BELIEFS

- UPDATE-DEFEAT-STATUSES, letting *altered-nodes* be the inference-nodes that are inference/defeat-descendants of the members of *new-links*.
- COMPUTE-UNDEFEATED-DEGREES-OF-SUPPORT, letting *new-beliefs* be the list of nodes whose undefeated-degrees-of-support (i.e., defeat-statuses) increase as a result of this computation, and *new-retractions* be the list of nodes whose undefeated-degrees-of-support decrease as a result of this computation.
- COMPUTE-INTEREST-GRAPH-DEFEAT-STATUSES using *new-beliefs and new-retractions*, letting *altered-interests* be the *interest-graph* nodes whose defeat-statuses change.
- For each support-link in *new-links*, if its support-link-target has no support-link not in *new-links*, insert the support-link-target into the *inference-queue*.
- For each member *S* of *new-beliefs*, APPLY-OPTATIVE-DISPOSITIONS-TO *S* and APPLY-Q&I-MODULES-TO *S*.
- DISCHARGE-ULTIMATE-EPISTEMIC-INTERESTS *new-beliefs new-retractions*, letting *altered-queries* be the list of queries whose query-answered-flags change value as a result.
- RECOMPUTE-PRIORITIES using *new-beliefs, new-retractions*, *altered-interests* and *altered-queries*, and reorder the *inference-queue*.

Optative dispositions are implemented as functions, and applying them just consists of applying the functions to *S*.

Applying Q&I modules is more complex. They function in much the same way as prima facie reasons, but there are some important differences. First, they are applied only to beliefs (nodes having nonzero undefeated-degrees-of-support), not to defeated nodes. Second, at least in human beings, it seems that the basis for a node drawn with a Q&I module is not recorded along with the node. As in the discussion in chapter two of *Cognitive Carpentry* of Q&I modules for induction, that is often part of the point of Q&I modules. It is unclear to me whether this is an essential feature of Q&I modules. Perhaps there could be different kinds. But for current purposes, I will assume that all Q&I modules work this way. So when a node is drawn by applying a Q&I module to a new belief, it will be recorded in an *inference-graph* node having an empty list of support-links, and the node-justification will be recorded as "Q&I". The reasoner should adopt interest in defeaters just as in the case of prima facie reasons, but there is again a complication. As the node-basis is empty, the undercutting defeater cannot be constructed as a formula of the form ⌜*basis* ⊗ *node*⌝. Instead we must simply stipulate that undercutting defeaters are of the form

> *P* is believed on the basis of a Q&I module under circumstances of type *C*, and the probability of *P* being true given that it is believed on the basis of a Q&I module under circumstances of type *C* is insufficiently high for the belief to be justified.

where *C* is projectible (see chapter two of *Cognitive Carpentry*). With this understanding, we have:

APPLY-Q&I-MODULES-TO *node*

Apply each Q&I module to *node* to construct inferrable sequents *S*, and then DRAW-CONCLUSION *S* ∅ "Q&I".

DISCHARGE-ULTIMATE-EPISTEMIC-INTERESTS *new-beliefs new-retractions*

- For each node *C* in *new-beliefs* and each query *Q* in *ultimate-epistemic-interests*, if *C* is in the list of query-answers for *Q* and the undefeated-degree-of-support of *C* is now greater than or equal to the degree-of-interest of query but was not previously:
  - Perform the positive-query-instruction of *Q* on *C*.
  - Set the query-answered-flag for *Q* to T.
- For each node *C* in *new-retractions* and query *Q* in *ultimate-epistemic-interests*, if *C* is in the list of query-answers for *Q* and the undefeated-degree-of-support of *C* was greater than or equal to the degree-of-interest of query but is no longer:
  - Perform the negative-query-instruction of *Q* on *C*.
  - If no members of the list of query-answers for *Q* are believed, set the query-answered-flag for *Q* to NIL.

RECOMPUTE-PRIORITIES *new-beliefs new-retractions altered-interests altered-queries*

Let *affected-nodes* and *affected-interests* be all the inference-nodes and interests whose discounted-node-strengths and interest-priorities are affected by changing the undefeated-degrees-of-support of the *new-beliefs* and *new-retractions*, the interest-priorities of *altered-interests*, and the answered-status of *altered-queries*. If any members of *affected-nodes* and *affected-interests* are still on the *inference-queue*, recompute their degrees of preference and reorder the *inference-queue*.

5. Initiating Action

In OSCAR's "doxastic" architecture for practical reasoning, action initiation begins with a permanent-ultimate-epistemic-interest in finding executable plan-nodes. When OSCAR's reasoning produces a belief that a certain node is executable and has no subsidiary-plan, the positive-query-instruction for the query initiating the search is executed. That query-instruction inserts a pair ⟨*A*,δ⟩ into the list of *executable-operations*. *A* is the term designating the operation, taken from the formula expressing the belief that the plan-node is executable. This term will have the form of a list whose first member is an operation type and the remaining members are parameters determining the specific character of the operation to be executed. For example, the operation might be of the type *move-your-index-finger*, and the parameters would specify a direction and distance. δ is a real number encoding the strength of the desire to perform the operation. *A* is assumed to be a basic action and it is executed by executing built-in subroutines. INITIATE-ACTIONS then operates on the list *executable-acts*. It chooses a maximally desired act, and tries to perform it:

INITIATE-ACTIONS

Let ⟨*A*,δ⟩ be the first member of *executable-operations* such that δ is greater than or equal to any γ such that for some *b*, ⟨*B*,γ⟩ is a member of *executable-operations*:

- Delete ⟨*A*,δ⟩ from the list of *executable-acts*.
- Try to perform the operation designated by *A*.
- Query epistemic cognition about whether *A* was successfully executed. Let the strength of the query be δ, the query-formula be ⌜*A* was not executed when the agent tried to execute it⌝, the positive-query-instruction be to reinsert ⟨*A*,δ⟩ into the list of *executable-operations*, and the negative--query-instruction be to delete ⟨*A*,δ⟩ from the list of *executable-operations*.

Trying to perform an executable act *A* relative to the null plan consists of performing some built-in routines that activate extensors. This will be implemented using a list *act-executors* consisting of pairs ⟨*act-type*,*routine*⟩. TRY-TO-PERFORM searches *act-executors* for a pair ⟨*act-type,routine*⟩ such that *act-type* matches the first member of *A*, and then it applies the function *routine* to the remaining members of *A*.

It is compatible with this architecture that pairs ⟨*A*,δ⟩ are inserted into the list of *executable-operations* in other ways as well. In particular, an agent may have built-in or conditioned reflexes that accomplish the same thing more quickly than planning in special cases.

Until OSCAR is supplied with reasoning schemes for reasoning about act executability and performance, these functions will not be called. This topic will be addressed in a later chapter.