Introduction to Cyc Inferencing

Overview of Cyc Inferencing

The Cyc inference engine handles modus ponens and modus tollens (contrapositive) inferencing, universal and existential quantification, and mathematical inferencing. It uses contexts called microtheories to optimize inferencing by restricting search domains.

The Cyc knowledge base contains over 1 million assertions. Many approaches commonly taken by other inference engines (such as frames, RETE match, Prolog, etc.) just don't scale well to KBs of this size. As a result, the Cyc team has been forced to develop other techniques.

Cyc also includes several special-purpose inferencing modules for handling a few specific classes of inference. One set of modules handles reasoning concerning collection membership, subsethood, and disjointness. Another handles equality reasoning. Others implement fast reasoning. Still others implement symmetry, transitivity and reflexivity reasoning.

The Internal Representation of Assertions

In previous versions of Cyc, formulas are stored and reasoned with in the same form in which they appear in the KB browser, e.g.

```
(implies
    (and
        (isa ?afp AdultFemalePerson)
        (residesInRegion ?afp Guam))
    (and (acquaintedWith Zippy ?afp)
        (likesAsFriend Zippy ?afp)))
```

In Cyc-10, formulas asserted to the KB are stored internally, and reasoned with, in conjunctive normal form (CNF). When converted to CNF, a formula gets rewritten as a conjunction of disjunctions of negated and non-negated literals. So, for example, the formula above would be written in CNF as:

```
(and
   (or
        (not (isa ?afp AdultFemalePerson))
        (not (residesInRegion ?afp Guam))
        (acquaintedWith Zippy ?afp))
   (or
        (not (isa ?afp AdultFemalePerson))
        (not (residesInRegion ?afp Guam))
        (likesAsFriend Zippy ?afp)))
```

Each of the conjuncts would become a separate assertion.

Converting to CNF is part of the job of the Cyc-10 canonicalizer. The canonicalizer turns CycL formulas into canonical form, so that they can be added to the KB as assertions, looked up in the KB, etc. Some of the other things the canonicalizer does are outlined below.

In Cyc-10, as well as in earlier versions of Cyc, universal quantification is handled trivially by leaving universally quantified variables unbound, while existential quantification is handled through the use of Skolem functions. Thus an assertion which is originally entered as:

```
(forAll ?A
    (implies
        (isa ?A Animal)
        (thereExists ?M
                (and
                        (mother ?A ?M)
                        (isa ?M FemaleAnimal)))))
```

will be converted to something like:

```
(implies
    (isa ?A Animal)
    (and
        (mother ?A (SkolemFunctionFn (?A) ?M))
        (isa (SkolemFunctionFn (?A) ?M)
             FemaleAnimal)))
```

and then further converted to CNF:

```
(and
    (or
        (not (isa ?A Animal))
        (mother ?A (SkolemFunctionFn (?A) ?M)))
    (or
        (not (isa ?A Animal))
        (isa (SkolemFunctionFn (?A) ?M)
             FemaleAnimal)))
```

Skolem functions are handled exactly like all other functions (except that Cyc creates and names the function term for you). You are free to rename the function term, or do anything else with it that you might do with a function such as `MotherOf`.

Another task the Cyc-10 canonicalizer takes care of is the ordering of literals in assertions. The advantage of a canonical literal order is that it simplifies KB lookup; only a test for structural equality is required to determine that 2 formulas are the same.

The major advantage to using CNF as an internal representation is that it greatly simplifies the conceptual scheme used in inferencing, because all axioms have a uniform structure. When you add

P and Q => R

to the system, it gets canonicalized to the CNF form

not(P) or not(Q) or R

Note that both of the following would be canonicalized to the same CNF form:

P and not(R) => not(Q)

Q and not(R) => not(P)

There is only one potential downside to using CNF, which is that certain types of assertions which can be expressed quite compactly in conditional form become somewhat unwieldy when converted to CNF. Specifically, an assertion of the form:

```
(implies
    (or  P1 P2 P3 ... Pm)
    (and Q1 Q2 Q3 ... Qn))
```

will result in a CNF with m times n conjuncts. However, we do not regard this as a significant problem, since asserting formulas of this form constitutes bad KE style, and thus is unlikely to occur very often. In particular, a knowledge enterer who writes such a formula is probably attempting to use the P1 ... Pm literals as an exhaustive list of cases in which the consequent should hold, when he or she should have been shooting for a single meaningful generalization.

Inferencing: An Introduction

Backward inferencing - the type of inferencing initiated by an ASK operation - can be regarded as a search through a tree of nodes, where each node represents a CycL formula for which bindings are sought, and each link represents a transformation achieved by employing an assertion in the knowledge base.

For example, let's say I ask for bindings for the formula `(likesObject ?x ?y)`. That formula will constitute the root node of an inference search tree. What I am looking for is any assertion which will help provide bindings for `?x` and `?y`. The KB may contain some ground assertions involving `likesObject`, such as

(likesObject Keith BillM)

and also some if-then rules, such as

```
(implies
    (possesses ?x ?y)
    (likesObject ?x ?y))
```

Each of these provides a way to expand the root node. That is, each constitutes a link to a new node with a different formula to satisfy; these new nodes will be the leaf nodes of the search. In the first case, the formula to satisfy in the new node is simply

#$True

In the second case, using the if-then rule takes us to a new node that now needs to satisfy the formula

(possesses ?x ?y)

The search procedure may now recurse on this new node, because if we can find bindings for this formula, the if-then rule will give us bindings for our original formula. Perhaps the KB also contains a rule which says,

```
(implies
        (objectFoundInLocation ?x KeithsHouse)
        (possesses Keith ?x))
```

This assertion can take us to yet another node with the goal formula

```
(objectFoundInLocation ?x KeithsHouse)
```

That is, we are now looking for things found in Keith's house, because if something is found in Keith's house, then Keith possesses it, and if someone possesses something, then he or she likes it. Note that this new node has one less free variable than its parent, which is probably desirable.

Alternatively, there may be another rule which states

```
(implies
        (and
                (isa ?x Agent)
                (owns ?x ?y))
        (possesses ?x ?y))
```

This rule could take us to a new node whose formula to satisfy is

```
(and
        (isa ?x Agent)
        (owns ?x ?y))
```

But this is probably not a happy development: we are now three nodes down, and the problem is getting more complex, rather than simpler.

Thus, the primary issue to be addressed in designing an inference procedure is the algorithm to be used for searching the tree. How do we decide which leaf nodes to expand next? Another important issue is to determine how a node is expanded. That is, how do we find the axioms in the knowledge base which are likely to provide links to new leaf nodes?

Inferencing in Cyc

Three important strengths of Cyc inferencing are:

1. The inferencing code is modular and stable.
2. The state of the search is maintained, so that a search which suspends due to resouce constraints can resume where it left off.
3. Search is completely parameterized, so that various kinds of search may be performed.

The first and second items should be self-explanatory, but the third demands elaboration. Cyc-10 does heuristic search by default, but depth-first search is also implemented (and is used for forward inference, since forward inference requires traversal of the whole search tree anyway). Certain applications also take advantage of parameterized search, doing mostly heuristic search but substituting in a special method for expanding nodes, or identifying goal nodes, etc. Below we discuss the heuristics Cyc-10 search uses and then we cover the various search parameters.

Heuristics for Deciding Which Node to Expand

Cyc-10 uses a number of heuristic rules to decide which leaf node to expand next. Some of these are purely syntactic heuristics:

- **Favor nodes with fewer literals to satisfy** (as compared with all other unexpanded nodes). This heuristic helps to steer the search toward branches that are likely to bottom out soon.
- **Favor nodes that have fewer free variables** (as compared with all other unexpanded nodes). Like the preceding heuristic, this heuristic helps to steer the search toward branches that are likely to bottom out soon.

- **Very weakly favor nodes that are higher up in the search tree** . This heuristic helps to avoid going too far down into the search without ever following up on other branches.
- **Strongly favor nodes with no free variables left at all** . If there are no free variables left, there's a good chance we'll be able to determine the truth value of the formula just by doing a KB lookup.
- **Weakly disfavor nodes which include negative literals** (that is, anything of the form `(not P)`). Because the KB consists primarily of positive assertions, it is easier to find bindings for positive literals than for negative literals.

Other heuristics are semantic:

- **Disfavor nodes which might be part of a unification cycle** . Basically, try to avoid going in circles.
- **Disfavor nodes which are less likely to be satisfied by KB lookup** . This is done by adding a measure of disfavor to a node for each of its goal literals that mention a predicate and a constant, but the constant has no index for the predicate in the KB.

Weighing Heuristics

These heuristic rules act in concert, according to a linear summation rule. One way to think of this is as a chorus of agents, each of which corresponds to one heuristic and who are each looking to see if their heuristic applies. As each candidate node is presented to the chorus, the agents shriek more or less loudly, according to how strongly they disfavor the node. The sum of the volumes of the shrieks is compared with the sums of other nodes, and the open node with the lowest sum is expanded next.

Heuristics for Deciding Which Literal in a Node to Expand

In many cases, the formula at a node will contain more than one literal. In the example developed above, for instance, we saw a node whose formula to satisfy was

```
(and
      (isa ?x Agent)
      (owns ?x ?y))
```

If we choose to expand this node, which of these two literals should we expand first? Should we look for elements of `Agent`, or should we look for `owns` pairs?

The solution used in Cyc-10 is to require that each HL module which applies to a literal must be able to provide a heuristic estimate for how expensive it would be to apply that module to the literal. These heuristics are linearly summed, and the literal which has the lowest heuristic cost is favored most strongly.

Search Nodes

Under Cyc-10, each search node is a structure with the following slots:

- search : the search this node is part of
- parent : this node's parent node
- children : A list of pointers to this node's children
- depth : the depth of this node in the search tree
- options : current ways left to expand the node
- state : a datastructure that contains the semantics of the node

The search node semantic state includes the following information:

- formula : the CycL formula to satisfy, which represents the intermediate state of the inference search at this node.
- inference supports : the assertions and HL modules used to transform our parent's formula into our formula. The inference supports of this node and all its ancestors together constitute the complete inference path down to this search node.
- variable bindings : a mapping between variables in the parent node and what they are bound to in this node.

ASKs and Direction

ASKs and ASSERTs use direction to control which assertions will be accessed during inference. Direction comes in two flavors: forward and backward. An ASSERT with direction :forward will cause inference to be performed at assert time; an ASSERT with direction :backward will cause inference to be deferred until ask time.

Equality

Equality is handled at unification time. It's as if the KB had a unique names assumption: objects with different names are assumed to be not equal, unless you specifically override the assumption by asserting that two objects are equal, e.g. `(equals Fred Joe)`. No inference concerning equality is done at unification time.