An Epistemological Level Interface for CYC

Mark Derthick
MCC
3500 West Balcones Center Drive
Austin, TX 78759
(512)338-3724
derthick@mcc.com

February, 1990

Abstract

The field of AI continues to be polarized by arguments between Neats, who advocate a top-down approach beginning with a clear formalization of the problem to be solved and develop broad theories, and the Scruffies, who advocate bottom-up heuristic programs and more often develop systems of commercial use. Following McCarthy and Hayes's [1969] suggestion for a distinction between *epistemological* and *heuristic* levels, we have added a logic-based interface to the CYC knowledge representation system, which makes it look neat. A translator converts between epistemological level and heuristic level representations. This uncouples the KB from the underlying heuristic reasoning system. If a desired conclusion is not drawn, the blame can be placed unambiguously on the knowledge enterer if the epistemological semantics do not justify it, or otherwise on the system implementor. Over the short term, at least, one does not have to depend on the other. As a matter of fact, CYC's implementors have satisfied its knowledge enterers over the long term. This paper describes the advantages of making this distinction, the user's view of CYC, the implementor's view, and how the translator keeps them in tune.

Contents

1 Introduction

The CYC Knowledge Base [Lenat and Guha, 1990] houses declarative representations of common-sense knowledge. Although the focus is on axiomatization, axioms without inference procedures are of little use for problem solving. Therefore much effort has of necessity gone into implementation of the supporting reasoning system, termed the *heuristic level* [McCarthy and Hayes, 1969]. The motivation for the heuristic level (HL) is pragmatic—it should find most of the desired conclusions quickly. Consequently, as CYC is applied to new domains, the HL evolves to accommodate the reasoning required. For instance it currently includes special facilities for increasing the efficiency of inheritance, inverse slots, temporal reasoning, part-whole structures, scripts, and higher-arity predicates. Driven by the need for efficient execution, the HL's reasoning is sufficiently *ad hoc* that it cannot be assigned a simple declarative semantics. Therefore in 1987 an *epistemological level* [McCarthy and Hayes, 1969] logic, called the Constraint Language, was defined that will remain relatively stable, and that has an intelligible semantics. The meaning of KB assertions is interpreted relative to this semantics. "Epistemological Level" and "Constraint Language Level" are used interchangeably.

At any given time, it is expected that the HL will be incomplete and/or unsound with respect to the constraint language semantics. That is, the HL will fail to derive certain valid conclusions, and will derive certain invalid conclusions.[1] Still, the clean separation between epistemological and heuristic levels divides the problems inherent in the expressiveness/tractability tradeoff between knowledge enterers, who may for the most part ignore implementation quirks, and the HL implementors, who have been largely successful in providing the inferences that are actually wanted (see figure 1). This separation has allowed rather drastic modification to the HL with only minimal disruption of work on the KB. For instance, in 1988 the HL used certainty factors, but in 1989 switched to a five-valued default logic. The Constraint Language has had a two-valued logic throughout [Guha, 1990].

Although the epistemological/heuristic distinction was proposed twenty years ago, and similar proposals have also been made [Newell, 1982, Levesque,

[1] In non-monotonic reasoning systems unsoundness usually goes hand-in-hand with incompleteness, because if you fail to derive an exceptional fact (Tweety is a penguin) you will incorrectly derive facts using the assertion it would have overridden (Tweety flies).

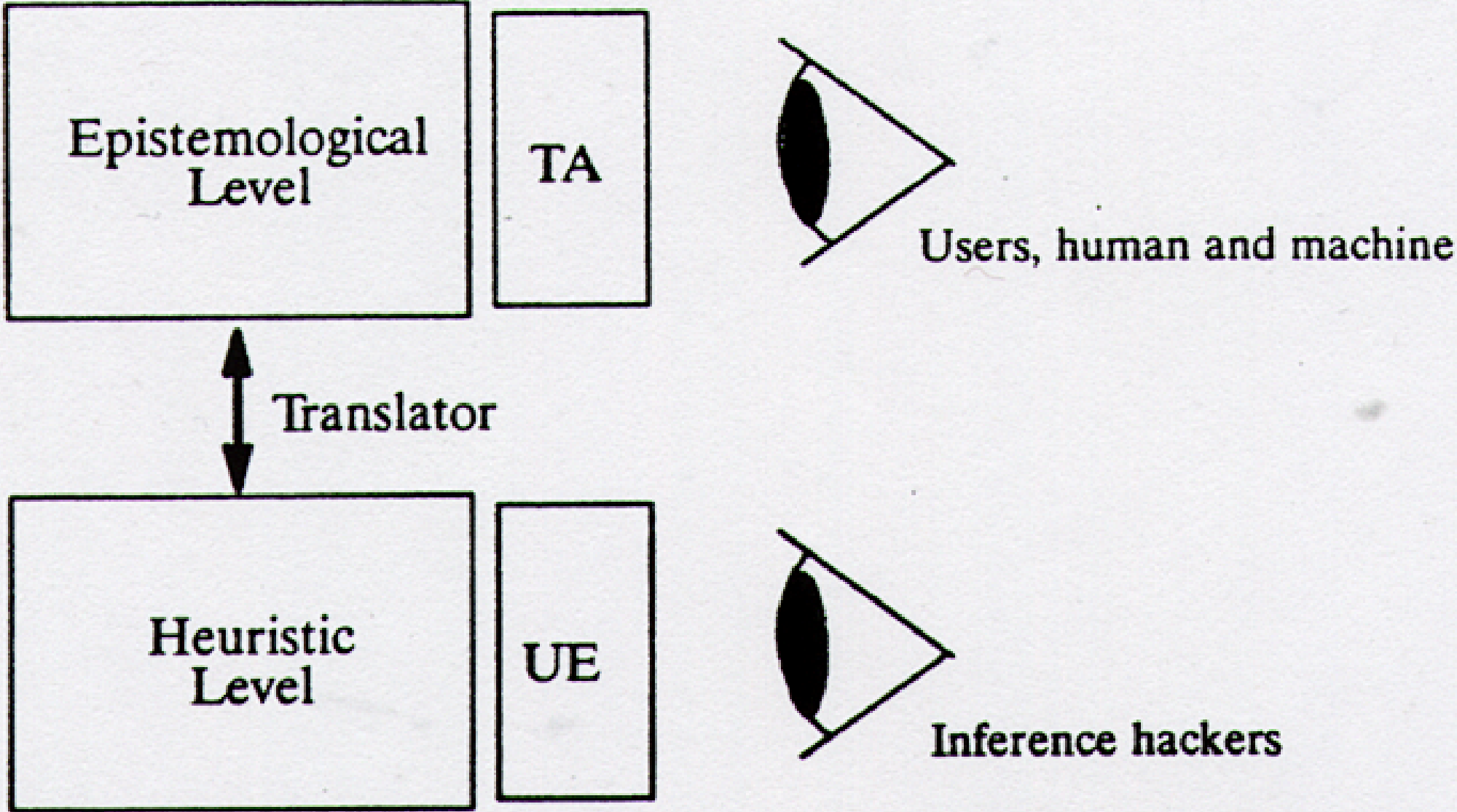
Epistemological Level
TA
Users, human and machine
Translator
Heuristic Level
UE
Inference hackers

Figure 1: Schematic representation of the relationship between the Epistemological Level and the Heuristic Level. All run-time reasoning takes place at the HL, but the translator can make this transparent, enabling the user to interact with CYC at either level. Currently human users primarily enter knowledge with the Unit Editor (UE) at the Heuristic Level, because the organization of knowledge into frames makes browsing easier. For other programs, and for humans entering complex constraints, the Tell/Ask (TA) interface, at the Epistemological Level, is more appropriate.

1984, Dennett, 1978], it has been used primarily to justify a divide and conquer approach at the level of schools of thought, rather than at the level of system modules: Neats worry about epistemology, and scruffies worry about heuristics. Few systems have consciously chosen to combine a well-defined expressive representation language with an incomplete and unsound heuristic inference algorithm. Indeed, at first sight, a well-defined semantics that doesn't quite fit the system's behavior seems pointless. However for a system attempting to represent all of common sense knowledge, the only alternative is declining to give a semantics independent of the program's behavior, because no expressively adequate formal language will be decidable. CYC's behavior is close enough to it's specification that (in Dennett's terms) taking an intentional stance is more productive than taking a design stance.

2 TA Overview

Levesque [1984] treats a KB as an abstract data type by defining its behavior in terms of two operations, Tell and Ask. In the same spirit, CYC's Tell/Ask interface, the TA, enables interaction with the system using the constraint language. Sentences are internally translated into the most efficient HL form automatically. The interface consists of four functions:[2]

Ask Extensionally takes a sentence, set expression, or term and returns the truth value, set, or individual based on the current KB. Alternatively, given a formula, it can return a list of bindings for the unbound variables that result in a true sentence.

Ask Intensionally takes a sentence and returns true if a proof can be found for the sentence.

For ground sentences these two functions behave identically, however for quantified sentences the result of one does not necessarily entail anything about the result of the other (due to the non-monotonicity of the constraint language). For instance if we ask whether all birds fly, and it happens that all currently known (to the KB) birds are known to fly, Ask Extensionally

[2] As briefly described in section 5.4, one of the reasons CYC requires more functions is that there is no explicit introspection operator.

will return true. Only if there is an axiom logically equivalent to "all birds fly" (not just those individuals that are currently known to be birds) will Ask Intensionally return true.[3] And if there is such an axiom, but some individual birds are known to be exceptions, then Ask Extensionally will return false, but Ask Intensionally will return true.

Justify Takes the same argument as Ask Intensionally, but returns the proof tree, rather than just "true," if a proof is found.

Tell adds an axiom to the KB such that the sentence will be (intensionally) true. Optionally it will create a unit to stand as a reified term for the sentence. This unit will have a slot holding the HL representation. So if the latter changes, due to incompatible changes in the code or availability of more efficient implementations, the Epistemological Level handle can remain constant. Conceptually there could be a Tell Extensionally also: Given "all birds fly," Tell Extensionally would assert of all currently known birds that they fly. This does not seem to be very useful, and it runs into problems for disjunctions and existentials [Levesque, 1984].

The bulk of this paper discusses the strategies used by Tell to find the most efficient frame language implementation of a constraint language proposition. The other three functions can be written easily given the modules built for Tell.

3 Constraint Language

The constraint language (CL) closely resembles predicate calculus, with the main difference being that quantifiers have explicit domains. That is, instead of $(\forall x)(P(x))$ we expect $(\forall x \in S)(P(x))$. This is written in the CL as (#%ForAll x S (P x)). We use explicit domains (S) for two reasons. First, in common-sense utterances, "every" and "some" are usually used with an explicit or implicit domain. It seems that this reflects that there are few useful common-sense generalizations that apply to everything. Second, it makes extensional reasoning much easier by drastically cutting down the number

[3] Ask Intensionally will not find proofs of more than one step except in one common special case described in section 5.1.1.

of bindings for the variables. Our notation for representing sets includes an unusual notation: (u s) indicates $\{x|(s\ u\ x)\}$. For example (#%Student #%allInstances) = $\{student|(\#\%allInstances\ \#\%Student\ student\}$ represents the set of all students. Of course one could always trivialize S as the universal set—(#%Thing #%allInstances), but in practice we rarely need to. The prefix "#%" denotes CYC frames, which are called *units*.

4 "Frame" Language

The HL is sometimes referred to as a *frame language* because historically it was almost exclusively based on frames, with each *unit* having a number of *slots* containing a number of *entries*. Slots correspond to two-place predicates in the constraint language, and the limitation to predicates of arity two or less proved sufficiently confining that arbitrary arity is now allowed. The important aspect of the HL from the point of view of the TA are the inference rules. In contrast to languages like Prolog, which uses only resolution on horn clauses, the HL does not rely on one or a few very powerful inference rules. Shielded from most users by the Epistemological Level, the HL is free to compile assertions idiosyncratically. The vast majority of assertions in the KB fall into one of a few dozen syntactic schemas for which inferences can be accomplished by special-purpose procedures that avoid the overhead of explicit variable binding, and that reduce the overhead required for truth maintenance in the general case by efficient indexing.

As an example of a HL inference rule, TransfersThro is described here. This rule can be used to infer "The owner of some thing also owns that thing's parts," for instance:

$$(\#\%ownedBy\ object\ owner) \wedge (\#\%physicalParts\ object\ part) \rightarrow (\#\%ownedBy\ part\ owner)$$

The general TransfersThro rule is $\dfrac{(s1\ x\ y) \wedge (s2\ x\ z) \rightarrow (s1\ z\ y) \quad (s1\ x\ y) \quad (s2\ x\ z)}{(s1\ z\ y)}$

where s1 and s2 are schema constants and x, y, and z are (universally quantified) variables. To make it easy to determine at run time when this rule applies, input sentence matching the schema in the top line of this rule are stored as $(\#\%transfersThro\ s1\ s2)$, and pointers to the special purpose

procedures implementing TransfersThro are associated with $s1$ and $s2$. Thus the job of the TA is produce (#%transfersThro #%ownedBy #%physicalParts) from a constraint language sentence like

```
(#%ForAll owner (#%Agent #%allInstances)
  (#%ForAll object (owner #%owns)
    (#%ForAll part (object #%physicalParts)
      (#%owns owner part))))
```

In general, deciding whether a sentence given to Tell matches the schema of an arbitrary inference rule is as hard as theorem proving. However most of the inference rule schemas actually used are simple enough that it is possible to canonicalize the input sentence so that matching requires time only linear in the length of the canonical form (see section 5.1.1). The more complex inference rule schemas involve sentential variables, and are treated differently (section 5.1.2).

Pragmatic information is also associated with assertions at the Heuristic Level, such as whether inferences using a TransfersThro assertion are to be drawn at assertion time or query time. Currently the TA ignores pragmatics, and the user is asked these questions by the HL directly. We plan to add implementation-independent resource specifications to the epistemological level interface, and to enable the translator to make educated guesses about this aspect of the HL implementation as well.

5 TA Implementation

5.1 Tell

When Tell is given a sentence, it first puts it in a canonical clausal form, and then compares this form to templates corresponding to each of the HL inference rules. Currently there is a static preference among inference rules, and the translator prefers the inference rule with the first matching template.

5.1.1 Canonical Form

Clausifying The first pass at canonicalizing an input sentence is putting it in clausal form, which is a good format for matching.[4] All the structural dependencies of the constraint language form have been incorporated into the form of the Skolem terms, leaving a flat structure that can be easily reordered for maximum canonicality. If the clausal form of the input to Tell contains multiple clauses, each is translated independently, except those that share Skolem functions.

Negative literals are segregated and called the *left hand side* (LHS), while positive literals are called the *right hand side* (RHS). This is by analogy with rule-based systems in which a right-hand-side conclusion may be drawn if all the left-hand-side antecedents hold.

Incorporating Domain Specific Knowledge After clausifying the input, tautological or contradictory literals and clauses are eliminated. In addition to domain-independent methods for recognizing these cases, the application can push functions on the special variables *tautology-functions* and *subsumption-functions*. The former should recognize tautological literals (such as (#%allInstances #%Thing x) in CYC), and the latter should recognize "necessary" entailments between literals (such as $(\#\%owns\ x\ y) \rightarrow (\#\%ownedBy\ y\ x)$ in CYC). The inverse relationship between #%owns and #%ownedBy is not strictly necessary, but it is often useful to assume certain types of contingent axioms will not be retracted. Thus when Telling a sentence S to a KB, we may sometimes actually add a simpler sentence S', where $KB \vdash S' \rightarrow S$.

Role Chains and Inheritance Paths The special-purpose HL inference rule Inheritance[5] is very heavily used, and optimized code for handling it

[4] Charniak and McDermott [1985] clearly explain clausification, including Skolemization.

[5] "Inheritance" has several meanings in AI. Inheritance in CYC is like propagation of property values down ISA links in many procedural-oriented knowledge representation systems. But like "role chains" in KL-ONE-like languages [Brachman and Schmolze, 1985] it can involve *any* type of link (slot). For instance an infinite set inherits infiniteness to its supersets; a finite set inherits finiteness to its subsets; Dan Quayle inherits sleepiness to his listeners; and Fred inherits a dislike of Fred to his neighbors.

is deeply embedded in the HL. The TA also takes pains to use inheritance whenever possible, and so includes role chains in its canonical form. After clausification, pairs of literals that can be glommed together using role chains are found. On the left-hand-side of a clause, pairs of the form $((s1\ u\ v1)\ (s2\ v1\ v2)\cdots)$ are changed to $(((s1\ s2)\ u\ v2)\cdots)$. Multiple clausess with identical LHS's, $((LHS\ ((s1\ u\ v1)))\ (LHS\ ((s2\ v1\ v2)))\cdots)$ become $((LHS\ (((s1\ s2)\ u\ v2)))\cdots)$.

These contractions can continue in a similar fashion to produce longer role chains. Some templates use role chains directly and others can be more efficient by using the HL inheritance syntax. But the most important use of finding role chains is that any of the inference rules can be combined with inheritance. For instance

```
(#%ForAll s (#%TangibleSubstanceSlot #%allInstances)
  (#%ForAll units-having-s ((s #%makesSenseFor) #%allInstances)
    (#%ForAll parts (units-having-s #%physicalParts)
      (#%ForAll entry (units-having-s s) (s parts entry)))))
```

can be implemented in the HL by inheriting a TransfersThro #%physicalParts assertion to all TangibleSubstanceSlots.

In general, one would want the TA to be able to translate epistemological level sentences into HL assertions that utilize *any* combination of inference rules. Being based on template matching, rather than chaining, this cannot be done in general with the current algorithm. Inheritance is an easy special case. Once in clausal form, every LHS literal whose predicate is a role chain and whose second argument is a (universally quantified) variable determines a possible inheritance path. Each such path is pulled out in turn, and the remainder of the clause matched against the templates. Any successes can be then be converted into inherited assertions.

Multiple Clauses As mentioned above, when an epistemological sentence produces multiple clauses, usually each is translated separately. There are two special cases, however, one from convenience and one from necessity.

If and Only If Forms Although clauses represent uni-directional implication, four of the current HL inference rules are inherently bi-directional: SlotValueEquals, ComputeByUnioning, ComputeByIntersecting, and Inverse.

Since any bidirectional assertion will have more than one clause, clause-by-clause translation can never match these inference rules. The TA searches for a set of clauses equivalent to $(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \leftrightarrow (q_1 \vee q_2 \vee \cdots \vee q_m)$ and replaces them with the special syntax `(IFF (p1 p2 ... pn) (q1 q2 ... qm))`. This is poor terminology since `IFF` isn't symmetric. The templates for the four bi-directional inference rules are also of this form, so it is still possible to translate clause-by-clause. However it is no longer possible to translate a bi-directional sentence into multiple uni-directional inference features.

Skolem Functions A sentence that generates multiple clauses that share Skolem functions cannot be translated clause by clause, or the requirement that both occurrences of the Skolem function must refer to the same thing would be lost. Consequently another special syntax is used to group together clauses that share Skolem functions: `(MULTIPLE <list of co-dependent clauses>)`.

Ordering In general, when there are no constraints on the order of literals in a clause, an exponential number of templates are required to do simple matching. For the actual templates used, however, literal ordering is not important. Either each side of the clause contains fewer than two literals, or the clause still matches the template no matter how it is ordered. Ordering predicate and function arguments is important, however. Atoms whose predicate is a slot, called *slot-preds*, can either appear as (s u v1) or, if the slot has an inverse, (s-inverse v1 u). The number of templates that would be required to match all possible argument orderings grows multiplicatively.

There are several idiosyncratic ordering rules, which collectively are very successful. Eleven inference rules require only one template, three require two, and one requires five. This overstates the effectiveness of the canonicalization for the more general inference rules, however, because translation by template matching breaks down. Most of the work for these is really done through escapes to complicated Lisp functions.

5.1.2 Sentential Form

Translating is easy for simple inference rules like TransfersThro, whose schema constants stand only for units and never for CL expressions. The desired

HL assertion follows immediately by substituting the schema constants into atomic assertions (see section 4). For more complicated inference rules, canonicalization by itself is insufficient. The expressions bound to sentential schema constants may contain variables that must be bound at run-time. Consequently, execution time varies greatly with the order in which constraints are checked, the extensional size of the domains of quantification, and the nesting structure of the quantifiers. For instance Sufficient Conditions takes a formula with free variables s, x, and y, and concludes $(s\ y\ y)$ for any three bindings that satisfy the formula.

When given the clausal canonical form for such a formula, the algorithm first finds the most restrictive domain of quantification for all other variables (besides s, x, and y), both universal and existential. This is rather tricky, and is best analyzed by cases. Consider the following general clausal form:

$$\begin{array}{l}(((p_{11}\ p_{12}\ \ldots\ p_{1l_1})\ (q_{11}\ q_{12}\ \ldots\ q_{1m_1}))\\ \ ((p_{21}\ p_{22}\ \ldots\ p_{2l_2})\ (q_{21}\ q_{22}\ \ldots\ q_{2m_2}))\\ \ \ldots\\ \ ((p_{n1}\ p_{n2}\ \ldots\ p_{nl_n})\ (q_{n1}\ q_{n2}\ \ldots\ q_{nm_n})))\end{array}$$

We seek a minimal domain of quantification for a variable, u, such that if u ranges only over this domain, rather than (#%Thing #%allInstances), the truth value is unaffected. (Each literal may be a function of u.) It is always possible to "fail safe" by finding a larger domain than is strictly necessary, at some possible cost in execution efficiency. Corresponding to each literal, p_{ij}, we postulate a set P_{ij} such that $\forall u((p_{ij}\ u) \rightarrow u \in P_{ij}))$. One of the crucial subroutines will try to find minimal sets P_{ij}.

First consider u to be universal. Thinking proceedurally, to extensionally determine the truth of a universally quantified sentence, we evaluate the body for every element in the domain, returning false if any individual falsifies the body, and true otherwise. If we can determine *a priori* that an individual will make the clauses true, we can eliminate it from the domain without changing the value we will return. This is the case if it makes all of the LHS's false. (We could eliminate an element if it made all of the RHS's true, too, but no useful way to do this has been found.) For each clause, we calculate a domain that excludes elements that must falsify the LHS of that clause. Then an overall domain is obtained by unioning these. Often the domain found will be exactly one of the P_{ij}, which will often satisfy $\forall u((p_{ij}\ u) \leftrightarrow u \in P_{ij}))$. In

this case the corresponding p_{ij} can be eliminated from all the RHS's it occurs in. For instance in the clause `((((#%likes u v1)) ((#%acquaintedWith u v1))))` the domain for v1 is (u #%likes). Over this domain, the LHS of the only clause is automatically satisfied, so there is no need to explicitly check it, and we can simplify the clausal form to `((NIL ((#%acquaintedWith u v1))))`.

Now consider u to be existential. Each element has the potential to verify the clauses. So we can eliminate any element that must make any of the clauses false. This holds for an element that must make all RHS literals false while all the LHS literals are true. Since we don't know how to determine that a literal must be true at this point, we give up (and return (#%Thing #%allInstances)) unless the LHS is empty, which is very often the case. The domain for all RHS literals must be unioned. We could then take the intersection over clauses, but instead the smallest is currently chosen. In the case of existentials, when the domain expression guarantees that the RHS will be satisfied it can be replaced by (#%True).

In summary,

$$(\#\%ForAll\ u\ (\#\%Thing\ \#\%allInstances)\ \Phi) \quad \leftrightarrow \quad (\#\%ForAll\ u\ \left(\bigcup_{j=1}^{n}\bigcap_{k=1}^{l_j} P_{jk}\right)\ \Phi)$$

$$(\#\%ThereExists\ u\ (\#\%Thing\ \#\%allInstances)\ \Phi) \quad \leftrightarrow \quad (\#\%ThereExists\ u\ \left(\bigcap_{j|l_j=0}\bigcup_{k=1}^{m_j} Q_{jk}\right)\ \Phi)$$

It is good to be able to take intersections of the intermediate domains, because it will only decrease the size of the resultant domain. However it is also better to minimize the run-time evaluation of the domain expressions. Hence we prefer to take set min's rather than intersections. If one possible domain is easily seen to be a subset of another it is chosen, so (#%Person #%allInstances) is preferred to (#%Animal #%allInstances) and (x #%likes) to (x #%acquaintedWith). If simple intensional reasoning fails, the current extensional sizes of the domains are actually computed and the smallest is chosen.

With unions the size of the set increases *and* the cost of computing it increases. Hence considerable effort goes into optimizing certain cases by not actually having to take the unions. One common case is where an intermediate domain is (#%Thing #%allInstances) because none of the literals

considered in finding a domain are functions of u. In this case we try to pull those literals outside the scope of the quantifier binding u, so we can avoid the union. Keeping track of this information is hairy. Since the canonical form is in CNF, it is easier to pull a whole clause out of a universal quantifier than to pull out the constant RHS literals (from different clauses) of an existential quantifier. Although the algorithm will opportunistically pull subexpressions out of the scope of both universal and existential quantifiers, it can only keep track of subexpressions that *must* be pulled out for the former. To handle the other case, back-translate is called twice for each input sentence. Once it is given the clausal form of the sentence, and the second time it is given the clausal form of the negation of the sentence. This changes universal quantifiers into existential ones and vice-versa, so if the structure of the canonical form one way makes it hard to optimize, the other form often works better. For sufficiently complicated sentences neither form will lead to optimal domains, but the record for the sentences actually asserted to CYC is reasonably good.

There are three cases for determining a minimal P_{ij} from a p_{ij}. A literal (#%likes #%Fred x) determines a domain (#%Fred #%likes), and a literal of the form (#%Member x <set expression>) determines a domain <set expression>. If these schemata do not apply, the domain must be determined using the predicate's argument types. For instance, since (#%argumentOneType #%owns #%Agent), one can form the domain (#%Agent #%allInstances) from (#%owns u z). One could also form (#%TheSetOf u (#%Agent #%allInstances) (#%owns u z)) to allow removing the literal from the clause, but "hiding" the literal in the domain makes further optimization impossible.

5.2 Ask Intensionally

Ask Intensionally is the closest thing CYC provides to a theorem prover, but it adheres to the general approach of treating lots of special cases and not providing completeness, in order to avoid the complexity of complete inference. Given a theorem to prove, the TA operates exactly backwards, using Tell to generate some possible sets of assertions from which the theorem would follow, then checking to see if any of the sets are in the KB.

5.3 Justify

When Ask Intensionally finds a translation for the sentence that unifies with the KB, it already has a set of assertions that justify that sentence. However these assertions may have been themselves derived, so the answer may be uninformative. The input sentence itself, after all, is a theorem of the KB, and it certainly justifies itself. So given a set of possibly derived justifications from Ask Intensionally, Justify then examines the HL data structure for each to determine whether it was derived, and if so, what it was derived from. This expansion continues until the set contains only axioms, at which point the tree ofjustifications is returned.

5.4 Ask Extensionally

Intuitively, Ask Extensionally answers based on what is currently known in the KB, rather than what can be proved will always be true (modulo exceptions and retractions). This corresponds to making a closed-world assumption about all positive and negative literals in the clausal form of a sentence (or in any form where negation has been pushed in as far as possible). Let K be a modal operator meaning "believes" [Levesque, 1984, Moore, 1985] where the intent is to have a perfectly rational introspective agent so that $K\Phi \equiv KB \vdash \Phi$. Since our logic is undecidable this is impossible, and the algorithm for computing K will be incorrect. We avoid most of the complications discussed by Levesque and Moore by restricting K to apply to positive or negative literals.

$$\text{Ask Extensionally}(\bigwedge_i(\bigvee_j p_{ij} \vee \bigvee_j \neg q_{ij})) \equiv \bigwedge_i(\bigvee_j Kp_{ij} \vee \bigvee_j K\neg q_{ij})$$

Operationalizing this, a Constraint Language expression is first optimized by putting it in canonical clausal form, using domain-specific knowledge to remove redundancies, and finding the best order for literals, slot-preds, and quantifier nesting (see section 5.1.2). The optimized form is then compiled into structurally isomorphic Lisp code. This is a simple matter of translating #%ForAll into #'every, #%ThereExists into #'some, #%LogAnd into #'and, #%LogOr into #'or, and literals into the appropriate lookup function.

6 Conclusion

Having a separate epistemological level is serving CYC well. Soon it will be possible to interact with the KB solely using constraint language expressions as arguments to the four interface functions described in the introduction: Tell, Ask Extensionally, Ask Intensionally, and Justify.[6] Converting to any interface based on the concepts of telling and asking in a language resembling predicate calculus should be easy, and we are in fact considering machine-machine dialog between CYC and HITS [Hollan *et al.*, 1988], DART/KIF [Genesereth and Singh, 1989], and other systems. The same general strategies for adding an epistemological level described here can be applied to other KR systems as well.

Having a propositional interface, several enhancements to CYC suggest themselves. Ask Intensionally hardly deserves to be called a theorem prover today, since it does no chaining. Hooking up an already existing theorem prover for it to optionally call on should be easy, even if it is written in something besides Lisp and runs on a different computer. Having a theorem-proving facility also allows an abduction algorithm to be called on to guess the answer to a problem, or to suggest plausible axioms to add to the KB. If the user were to Tell the KB that all birds fly, for instance, and the KB knows that all birds have wings, Tell could offer as an option the rule that all winged things fly. If Tell is much better than knowledge enterers at choosing internal representations, a background task can be run that translates existing HL assertions using the more expensive inference rules into the CL, and offers these expressions to Tell. If Tell prefers a different implementation, the KB can be upgraded automatically. Even if Tell is not much better than knowledge enterers it can suggest changes to be approved by a person. Overall it seems that the TA is doing a better job than was being done before there was a separate epistemological level, because users often had a poor understanding of the trade-offs between the various inference rules. Typically they would learn a few and ignore the rest. Now users do not even have to be aware of their existence, and programs using CYC can now take advantage of HL features created *after* the program was written.

[6] The full CYC Functional Interface [Pratt, 1990] is a superset of these functions.

Acknowledgements

I thank R. V. Guha, Doug Lenat, and the other members of the CYC project for their suggestions in implementing the TA and for improving the content of this paper.

References

[Brachman and Schmolze, 1985] Ronald J. Brachman and James G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, March 1985.

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

[Dennett, 1978] Daniel C. Dennett. Intentional systems. In *Brainstorms*, chapter 1, pages 3–22. MIT Press, Cambridge, MA, 1978.

[Genesereth and Singh, 1989] Michael R. Genesereth and Narinder P. Singh. Knowledge interchange format. Technical Report Logic-89-13, Stanford University, October 1989.

[Guha, 1990] R. V. Guha. The representation of defaults in CYC. Technical Report ACT-CYC-083-90, MCC, February 1990.

[Hollan *et al.*, 1988] James Hollan, Elaine Rich, William Hill, David Wroblewski, Wayne Wilner, Kent Wittenburg, Jonathan Grudin, and Members of the Human Interface Laboratory. An introduction to HITS: Human interface tool suite. Technical Report ACT-HI-406-88, MCC, December 1988.

[Lenat and Guha, 1990] Douglas B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems*. Addison-Wesley, Reading, MA, 1990.

[Levesque, 1984] Hector J. Levesque. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23:155–212, 1984.

[McCarthy and Hayes, 1969] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[Moore, 1985] Robert C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[Newell, 1982] Allen Newell. The knowledge level. *Artificial Intelligence*, 18(1):87–127, 1982.

[Pratt, 1990] Dexter Pratt. A functional interface for CYC. Technical Report ACT-CYC-089-90, MCC, March 1990.