CS 289 Knowledge representation and reasoning
Fall 1995 Stuart Russell

Project 2

Reasoning facilities

The overall aim of the project is to gain "vertically integrated" experience with logical reasoning by implementing a complete, albeit simple, system from the ground up. This project should be done in pairs. Since the system you will build should interact with the user, it is easiest to build it as a set of lisp functions that the user can call directly. For an example of the this style of interaction, see the Epikit Introduction.

The logical language you will implement consists of atomic sentences and definite clauses, with all variables assumed universally quantified. Function symbols are allowed, as well as equality. You will need some syntactic method to distinguish constants from variables. You may find it convenient to make your sentence syntax consistent with Epikit's—in particular, the left-arrow for implication used with backward chaining. (This assignment uses the book's infix notation only for clarity.)

Carry out each of the following steps, making you sure you test each thoroughly before proceeding.

1. Implement data types, with appropriate access functions, for logical expressions and substitutions. (Remember that an empty substitution needs to be distinct from the result of a failed inference!)

2. Implement a unification routine that standardizes apart its arguments (traditionally, by renaming the variables of the second argument), and a substitution routine that returns the result of plugging a binding list into an expression.

3. Implement a simple mechanism for storing and retrieving facts in a knowledge base. SAVE(KB,p) saves sentence p in the KB (which is side-effected). KNOWNP(KB,p) is true if a sentence unifying with p is in the KB. KNOWNX and KNOWNS behave as in Epikit (but note the extra KB argument). DROP(KB,p) removes all facts matching p, and EMPTY(KB) drops the lot.

4. Implement the backward chaining algorithm from Chapter 9, with interface functions ASKP ASKX and ASKS.

5. Use your backward chainer to solve the inference problems in Exercise 7.6.

6. Given an atomic goal p, your ASK routines try both retrieval and chaining. Now modify it to allow for procedural attachment, so that if a sentence $ToAsk\{p/x/s\}(a, p')$ appears in the KB, where p' and p unify, then the function a is called with p as its argument. Make sure the function returns the appropriate information so that the result can be used just as if inference had taken place. Use addition as an example to show your modifications behave properly.

7. Make similar modifications to SAVE and the KNOWN routines. Use the example of storing chess positions in an array rather than as a collections of sentences.

8. Write a simple template-based natural language output facility NLG that translates logical sentences into "English." For example, the sentence $Age(x, y)$ could be translated into the sentence "x is y years old." The easiest way to do this is to store the templates in a knowledge base. For example,[1]

   $$Template(Age(x, y), [x, Is, y, Years, Old])$$

   (If no template matches the logical sentence, then it is printed in the standard syntax.) Whenever one of your routines wants to print a sentence, it should check to see if there is a *ToPrint* fact in the KB that specifies a function to be used; NLG is the default.

   Define a procedural attachment function USER that solves a goal by printing the goal out in "natural language" and asks the user to provide the solution.

9. An *explanation facility* allows the user to ask why the system reached certain conclusions. For example, if the KB contains p and $p \Rightarrow q$, and answers yes to the query q, the user should be able to ask "Why?". The system should reply something like "By Modus Ponens from (1) p and (2) $p \Rightarrow q$" perhaps with appropriate natural language generation thrown in. The user could then query 1 or 2, and so on. To implement this facility, you will need to store *justifications* for each successful inference step as well as all the intermediate conclusions. (It is probably a good idea to have a `*justify*` flag that is either `nil` or gives the KB into which justifications are to be stored.) Show your system explaining why George is Zara's greatgrandparent.

[1] Notice that sentences are being treated as terms here. Templates and procedural attachment are really metalevel information. From the point of view of the metalevel, the object-level sentence being reasoned about is really just a string. For this reason, make sure that any variables it contains are kept distinct from the metalevel variables.