A Case Study of Knowledge Representation in UCf

David N. Chin

Division of Computer Science
Department of EECS
University of California, Berkeley
Berkeley, CA. 94720

ABSTRACT

The knowledge representation used in UC provides a flexible framework suitable for a large variety of natural language processing tasks including parsing, inference, planning, goal analysis, and generation. Although many of the knowledge structures are specific to the UNIX Consultant domain, a common design goal is the use of associative processing. By providing direct links between related knowledge structures, inference and other processing can be done very efficiently. Access to representations in UC is by hash indexing which simulates a real associative memory.

1. Introduction

UC, the UNIX* Consultant, is a natural language program that converses in English with users in the domain of the UNIX operating system. UC provides information on usage of system utilities, UNIX terminology, and plans for accomplishing specific tasks in the UNIX environment, all upon direct query by the user. In order to accomplish these tasks, *UC* must perforce have a considerable knowledge base, a large part of which is particular to the UNIX domain. The specific representations used in this knowledge base are essential to the successful operation of UC. Not only are the knowledge structures used in parsing, inference, planning, goal detection, and generation, but also the format of representation must permit the high efficiency in access and processing of the knowledge that is required in an interactive system like UC. This paper describes the details of this representation scheme and how it manages to satisfy these goals of usability and efficiency. An overview of the UC system can be found in Wilensky (1982). For an introduction to the knowledge structures in UC, see Chin (1983). In contrast, a totally different approach to the same problem domain is described in Douglass and Hegner (1982).

S. The Representation

UC uses a framelike representation where some of the contents are based on Schank's conceptual dependencies. The knowledge structures are stored in PEARL databases (PEARL is an AI package developed at Berkeley that provides efficient access to Lisp representations through hashing mechanisms, c.f. Deering, et. al., 1981 and 1982). Since the knowledge to be represented is highly domain dependent, it is instructive to look at the range of queries that may be encountered by a UNIX consultant. The following dialogue is taken from actual UC sessions:

User: How can I delete a directory?

UC: To delete a directory with name directory name, type ' rm directoryname '.

User: What is a directory?

UC: *A* directory is kind of file which is used to hold a set of files.

User: Why can't I remove the directory Tmp?

UC: The directory Tmp must be empty before the directory can be deleted.

The first two queries are representative of direct requests for information from *UC* and differ only in the type of information and the method of processing. The first asks for the name and usage of a UNIX utility. In general, with queries of the first type, the user states the goals or results (hat are desired or the actions to be performed and then asks *UC* for a plan to achieve those goals or perform those actions in UNIX. To process such "how" questions, UC encodes in its data base a large number of plans or equivalently the knowledge necessary to generate those plans as needed. The serond query is an information request for the definition of certain UNIX or general operating systems terminology. Although such requests can be bandied easily by canned textual responses, it is much harder to encode the knowledge in a format useful for generation.

In the third query, the user describes a situation where his expectations have failed to be substantiated and asks *UC* to explain why. In general, the user simply presents a problem and asks for the solution. UC may have to infer the goals of the user and from that which plan the user likely had bern using. Once the plan is known, then UC must check for the proper execution of the plan, looking for misplaced steps or violated preconditions to the plan.

S. An Analysis

To see how the knowledge structures are actually used, it is instructive to follow the processing of queries in some detail. The English input is first parsed from left to right by the Phran parser (Wilensky and Arena, 1980a and b) into internal representations. For instance, the query of example three is parsed in sections. The clause, "Delete the directory Tmp" is represented as a causation frame where the user (actually a frame which is not shown) does something to cause the directory (another frame not shown) to no longer exist.

```
(causation (antecedent (do (actor *user*)))
           (consequent (state-change (actor directory 3)
                                     (state-name physical-state)
                                     (from existing)
                                     (to non-existing))))
```

The final form of the representation of query three is a question frame with the single slot, cd, which is filled by a causation frame. The causation represents "why" (with antecedent 'unknown*) the user is unable to perform the action, actl, which is the causation frame for deleting shown above.

```
(question (cd (causation (antecedent 'unknown*)
                         (consequent (able (actor *user*)
                                           (value unable)
                                           (perform actl))))))
```

f This research WAS SPONSORED it part by the Office of Naval Raearck under contract N000M-8O-C-0732 aad the National Science Foundations under fraat 1ST-8007045

• UNIX is trademark of Bell Laboratories

Once the input is parsed, UC which is a data driven program looks in its data base to find out what to do with the representation of the input. An assertion frame would normally result in additions to the database and an Imperative might result in actions (depending on the goal analysis). In this case, when UC sees a question about why someone is unable to do something, it interprets this as a statement that the user failed to perform the action, and that it should check the preconditions for the plan for that action. This knowledge is all encoded associatively in a memory-association frame where the recall-key is the associative component and the cluster slot contains a set of structures which are associated with the structure in the recall-key slot.

```
(memory-association
    (recall-key (question
                  (cd (causation
                        (antecedent 'unknown*)
                        (consequent (able (actor 'person)
                                          (value unable)
                                          (perform Tact)))))))
    (cluster ((fail (cd Tact))
              (goal (planner ?person)
                    (objective ?states))
              (assertion (cd (planfor (result ?states)
                                      (method ?plan))))
              (check-preconds (cd ?plan)))))
```

The above memory-association circumvents the need to infer that the user failed, to delete the directory, that the user has a goal and there is a plan carrying out the objectives of that goal. In UC, knowledge about plans *is* encoded in a large number of planfor frames which are used almost directly to handle informational queries about UNIX commands. In this example the usage of the plan is less direct since UC must check the preconditions for that plan. This intention is stored in the check-preconds frame.

The plan itself was actually activated earlier when "deleting the directory Tmp" was parsed and a memory-association *reminded* UC of the following planfor deleting the directory, Tmp, which was instantiated by unification with the patterns in the general planfor deleting directories:

```
(planfor (result (state-change (actor directory3)
                               (state-name physical-state)
                               (from existing)
                               (to non-existing)))
         (method (mtrans (actor *user*)
                         (object (command (name rmdir)
                                          (args (Tmp))
                                          (input *stdin*)
                                          (output *stdout*)
                                          (diagnostic *stdout*)))
                         (from *user*)
                         (to *Unix*)))))
```

When UC processes the check-preconds, it looks for a preconds frame for the given plan and checks the preconditions listed therein:

```
(preconds (plan (mtrans (actor Tuser)
                        (object (command (name rmdir)
                                         (args (?directoryname))
                                         (input *stdin*)
                                         (output *stdout*)
                                         (diagnostic *stdout*)))
                        (from ?user)
                        (to *Unix*)))
          (are ((state (actor (all (var ?file)
                                   (desc (file))
                                   (pred (inside-of
                                           (object 'directory)))))
                       (state-name physical-state)
                       (value non-existing))
```

One precondition in this case is that the directory must be empty before it can be deleted. Upon querying UNIX directly, UC determines that this precondition is not satisfied, so it generates a message using the Phred generator (Jacobs, 1083) to inform the user of this fact. Of course if this precondition was not the problem, UC would continue to check the rest. Also in multi-step plans, UC would make sure that the steps were carried out in the proper order.

The informational query of "How can I delete a directory" is handled in an even more straight forward way. It is first parsed into a question frame:

```
(question
    (cd (planfor (result (state-change (actor directory4)
                                       (state-name physical-6tate)
                                       (from existing)
                                       (to non-existing)))
                 (method 'unknown*))))
```

Then the following memory-association directs UC to look for an out-planfor frame whenever a question about how to do something (expressed as a planfor with an unknown method) is encountered:

```
(memory-association
    (recall-key (question (cd (planfor (result Tconc)
                                       (method *unknown*)))))
    (cluster ((out-planfor (query Tconc)
                           (plan ?*any*)))))
```

The meaning of an out-planfor is that an answer to the query in the query slot is to simply execute the plan in the plan slot. Theoretically outrplanfors represent compiled answers to queries which the expert consultant has encountered so often that an immediate solution is already available.

```
(out-planfor (query (state-change (actor ^directory)
                                  (state-name physical-state)
                                  (from existing)
                                  (to non-existing)))
             (plan (output (cd (planforIO)))))
```

In this case, the plan in the out-planfor is an output frame, the contents of which are passed to the generator. PlanforIO in the output frame is the general version of the planfor shown in the previous discussion. In more complex queries where UC does not have an out-planfor compiled away, the problem of creating a plan to handle that particular situation is done by the planning component of UC, Pandora (Faletti, 1082).

The processing of queries about terminology requires a somewhat different approach. Definitions are actually generated from basic knowledge. For example, the definition of a directory uses the fact that a directory is a kind of file in the type hierarchy of frames in UC. Also, used is the fact that a directory is a kind of functional-object all of which have *function*. The function of directories is to hold files. Although it would be possible for a consultant like UC to have definitions "compiled" away in its memory, it was felt that requests for definitions are so rare that this was unlikely, so UC generates all definitions from scratch.

4. Associative Processing for Efficiency

A common theme in the knowledge representation of UC is the use of associative processing to achieve efficiency. Instead of potentially exponential cost inference engines, the knowledge structures in UC are designed to provide direct associations among relevant knowledge. For example, planfors provide immediate links between plans and their effects, out-planfors connect common queries to their generator ready outputs, and memory-associations provide a general mechanism for associative links of almost any kind.

Although computational methods of inference are undoubted still needed and desired, the premise taken in UC is that most common everyday inferences made by humans are not computational but rather fall out of the structure of memory. This can be modeled in AI programs like UC through appropriate design of knowledge representations.

Lacking true associative memory, UC uses the hash indexing provided in PEARL databases to simulate associative access. Frames stored in PEARL databases are indexed by combinations of the frame type and/or the contents of selected slots. For instance, the planter of the example is indexed using a hash key based on the state-change in the planter's result slot. It is stored by the fact that it is a planter for the state-change of a directory's physical-state. This degree of detail in the indexing scheme allows this planter to be immediately recovered whenever a reference is made to deleting a directory.

Similarly, a memory-association is indexed by the filler of the recall-key slot, an out-plan for is indexed using the contents of the query slot of the out-planter, and a preconds is indexed by the plan in the plan slot of the preconds. Indeed all knowledge structures in UC have associated with them one or more indexing schemes which specify how to generate hashing keys for storage of the knowledge structure in the UC databases. These indexing methods are specified at the time that the knowledge structures are defined. Thus although care must be taken to choose good indexing schemes when defining the structure of a frame, the indexing scheme is used automatically whenever another instance of the frame is added to the UC databases. Also, even though the indexing schemes for large structures like planters involve many levels of embedded slots and frames, simpler knowledge structures usually have simpler indexing schemes. For example, the representation for users in UC are stored in two ways: by the fact that they are users and have a specific account name, and by the fact that they are users and have some given real name. Thus although extra work must be done to work out usable and efficient hash indexing schemes for structures, the gain in processing efficiency is well worthwhile.

6. Technical Data

UC is a working system which is still under development. In size, UC is currently two and a half megabytes of which half a megabyte is FRANZ lisp. Since the knowledge base is still growing, it is uncertain how much of an impact even more knowledge will have on the system especially when the program becomes too large to fit in main memory. In efficiency, queries to UC take between two and seven seconds of CPU time on a VAX 11/780. Currently, all the knowledge in UC is hand coded, however efforts are under way to automate the process.

6. Acknowledgments

Some of the knowledge structures used in UC are refinements of formats developed by Joe Faletti and Peter Norvig. Yigal Arens is responsible for the underlying memory structure used in UC (Arens, 1082) and of course, this project would not be possible without the guidance and advice of Robert Wilensky.

7. References

Arens, Y. 1982. The Context Model: Language Understanding in Context. In the *Proceedings of the Fourth Annual Conference of the Cognitive Science Society.* Ann Arbor, MI. August 1982.

Chin, D. 1983. Knowledge Structures in UC, the UNIX Consultant. In the *Proceedings of the 21 st Annual Meeting of the Association for Computational Linguistics.* Boston, MA. June, 1983.

Deering, M., J. Faletti, and R. Wilensky. 1981. PEARL: An Efficient Language for Artificial Intelligence Programming. In the *Proceedings of the Seventh International Joint Conference on Artificial Intelligence.* Vancouver, British Columbia. August, 1981.

Deering, M., J. Faletti, and R. Wilensky. 1982. The PEARL Users Manual. Berkeley Electronic Research Laboratory Memorandum No. UCB/ERL/M82/19. March, 1982.

Douglass, R., and S. Hegner. 1982. An Expert Consultant for the Unix System: Bridging the Gap Between the User and Command Language Semantics. In the *Proceedings of the Fourth National Conference of Canadian Society for Computational Studies of Intelligence.* Uriversity of Saskatchewan, Saskatoon, Canada.

Faletti, J. 1982. PANDORA - A Program for Doing Coromonsen&e Planning in Complex Situations. In the *Proceedings of the National Conference on Artificial Intelligence.* Pittsburgh, PA. August, 1982.

Jacobs, P. 1983 Generation in a Natural Language Interface. In the *Proceedings of the Eighth International Joint Conference on Artificial Intelligence.* Karlsruhe, Germany. August, 1983.

Wilensky, R 1982. Talking to UNIX in English: An Overview of UC. In the *Proceedings of the National Conference on Artificial Intelligence.* Pittsburgh, PA. August, 1982.

Wilensky, R. 1981(b). A Knowledge-based Approach to Natural Language Processing: A Progress Report. In the *Proceedings of the Seventh International Joint Conference on Artificial Intelligence.* Vancouver, British Columbia. August, 1981.

Wilensky, R., and Arens, Y. 1980(a). PHRAN - a Knowledge-Based Natural Language Undcrstander. In the *Proceedings of the 18th Annual Meeting of the Association for Computational Linguistics.* Philadelphia, PA.

Wilensky, R., and Arens, Y. 1980(b). PHRAN - a Knowledge Based Approach to Natural Language Analysis. University of California at Berkeley. Electronic Research Laboratory Memorandum No. UCB/ERL M80/34.