PEARL — A Package for Efficient Access to Representations in LISP

Michael Deering
Joseph Faletti
Robert Wilensky

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

Abstract

PEARL is an A] language developed with space and time efficiencies in mind. In addition to providing the usual facilities such as slot-filler objects, demons and associative data bases, PEARL introduces stronger typing on slots, user-assisted hashing mechanisms, and a forest of data bases. The resulting product is a simple, but powerful and efficient tool for AI research.

I Introduction

We have developed an AI language called PEARL (Package for Efficient Access to Representations in LISP). Unlike the recent tendency toward elaborate representation languages such as KRL [I] or language generators such as RRL [2], PEARL is a more modest system that combines a number of useful AI techniques into a very efficient package. Besides the usual goal of providing the user with a more meaningful interface than is available via LISP, PEARL has the following salient characteristics:

(1) PEARL combines some features of predicate calculus-like data bases with those of frame-based systems like FRL [3].

(2) PEARL introduces typing to user-defined knowledge structures.

(3) PEARL allows the user to manipulate a forest of associative data bases.

(4) Fetches from the data base return streams of answers. Retrieval is based on pattern matching.

(5) PEARL is very efficient. PEARL uses its own internal representation for knowledge structures for both economy of storage and speed. A great deal of effort has gone into exploiting type information not available in most AI languages to eliminate searching inefficiencies. In addition, the user may easily specify, as part of a knowledge structure definition, a great deal of information about how objects should be indexed for efficient retrieval. Thus PEARL provides much of the power of expression of other AI languages without the usual overhead.

Perhaps most significantly, PEARL is actually being used in the construction of several AI systems. In particular, the latest version of PAM [5] a story understanding program, has been re-programmed in PEARL. PANDORA [6] a planning program now under development at Berkeley, is also written in PEARL Our experience has led us to conclude that PEARL is an effective AI tool for the creation of efficient AI programs.

PEARL provides the user with a set of operators for defining, creating, and manipulating slot-filler objects, as well as placing them into associative data bases, upon which further operations may be performed.

II Objects and Structures

PEARL has four basic types: INTEGER. SYMBOL. STRUCTURE, and LISP. Objects of type INTEGER are the usual numeric type, and objects of type LISP can be any general LISP object. Symbols (objects of type SYMBOL) correspond to atoms in LISP, and are simply primitive objects with unique labels. Structures are collections of slots, each of which may be filled with objects of any (predeclared) type. There is also a meta-type. SETOF, which can be applied (recursively) to any basic type to generate a new type, which will consist of a list of the specified type of objects.

Types of structures must be predefined, with the number of slots, and their names and types specified via a user declaration. When an instance of a structure is created and its slots filled, only objects with the same type as the slot may fill it. In addition, new structures may build upon old ones in a hierarchical fashion by specifIng new slots to add to the old ones. This hierarchy may be used in operations upon the data base.

For example, symbols are declared as follows:

```
(SYMBOL John Home Office)
```

The statement:

```
(CREATE BASE Act
        (Actor      SYMBOL) )
```

will define the primitive type Act. More specific forms of Acts can now be defined in terms of this base, or in terms of other defined types:

```
(CREATE EXPANDED Act Trans
        (From       SYMBOL)
        (To         SYMBOL))
(CREATE EXPANDED Trans PTrans
        (Object     SYMBOL))
(CREATE EXPANDED Trans MTrans
        (MObject    STRUCT))
```

Instances of these structures can now be created as follows:

```
(CREATE INDIVIDUAL Ptrans
        (Actor      John)
        (Object     John)
        (From       Office)
        (To         Home))
```

This last structure denotes "John went home from the office" in Conceptual Dependency [4].

PEARL provides functions for accessing and changing the values of slots within individual structures, for automatically naming the structure created, and for defining other sorts of entities as well. Slots within a structure may also be filled with a pattern-matching variable, in which case the structure may be viewed as a pattern. Such variables are bound as part of the matching process (usually during a fetch from the data base.)

III Data Base Facilities

PEARL allows for a forest of associative data bases implemented as hash tables into which structures may be placed, and later fetched via structure patterns. A common problem with most AI data base implementations is the system's lack of knowledge about how best to organize information for efficient retrieval. Since the user usually has such knowledge. PEARL encourages the user to provide it when a structure type is defined to the system in the form of labels on those slots most likely to help with hashing the structure (that is. with distinguishing instances from one other).

For example, to indicate that a particular slot is useful for hashing, the user puts an • in that slot in the declaration. Thus

```
(CREATE BASE PlanFor
        (• Goal     STRUCT)
        (Plan       STRUCT))
```

defines a type PlanFor with slots for a goal and a plan, and indicates that PlanFors should be indexed to be retrieved by their goals.

In addition, we may also wish to specify how the value that fills the Goal slot is used to create the index. For example, if the Planner of the Goal is deemed significant for such purposes, we can indicate this as follows:

```
(CREATE BASE Goal
        (& Planner  SYMBOL)
        (Objective  STRUCT))
```

This will inform PEARL that structures that index on slots filled with Goal-type structures should use the Planner slot for further discriminations, however, the way in which Goals themselves are indexed will not be affected.

IV The Matcher

To fetch an object from a data base, the user invokes the fetcher with a structure to be used as a pattern. This pattern is matched against likely objects in the data base, and those which match are returned to the user via a result stream (actually possibilities list). The semantics of the matching have been constrained to avoid the usual variable naming problems for efficiency.

Two structures match if they can be unified, although no attempt is made to detect circularities (in our applications, these never occur). Also, most variables are local to the pattern they appear in. so naming conflicts do not arise (but see below for special kinds of variables). Predicates may also be attached to a slot specifying matching constraints.

There also is a general "if-added" demon mechanism, in which demons can be attached either to structure definitions (that is, to all instances of a particular type of structure) or to individual slots of individual instances of structures. These demons must be labelled with the function which they wish to monitor and are only invoked by PEARL when that function is being performed.

V Variables

There are three types of pattern matching variables in PEARL: global variables (which are just LISP variables), variables which are local to an individual structure and an intermediate type of variable which provides lexical scoping within groups of structures. Local variables are dummy variables, local to a particular structure and any of its components which were created in the same instant. They are all unbound before every match on that structure. Lexically scoped variables are like local variables in that they are unbound before a match is made, but have their scope extended across several structures as indicated by the user.

For example, a body of PEARL structures conceptually composing a single frame can be made to share the same variables. These variables will be local to that frame. However, the results of matching any particular component of the frame will be detectable in the variables associated with the other components.

VI Implementation Goals

While the main emphasis of efficiency considerations within PEARL was to allow the user to avoid exponential algorithms, we also tried to make the code itself as efficient as possible. To make the user interface as friendly as possible, error checking is done whenever it

can be done efficiently. As a result of these two principles. PEARL is fast and friendly enough for use both as a serious programming language as well as a teaching tool for graduate A] classes. PEARL was also intended to be portable. It was developed on a DEC 20. moved with no modification to a DEC PDP-10 under UC1 LISP, and currently also runs on a VAX-11/780 under Franz Lisp.

VII Implementation and Performance

PEARL achieves its space efficiency and some of its time efficiency by requesting a block of memory from LISP for each structure instance or definition. The contents or defining information are then packed within this block. Since much of the defining information is Boolean, this provides substantial savings in space for definitions. Data bases are managed similarly.

Only the lowest level functions for accessing these blocks could not be written in LISP. In the PDP-10 implementation, these functions are written in assembler; on the VAX they are written in C. The major effort required to move PEARL to another machine is to rewrite these functions. The rest of the job entails dealing with different dialects of LISP.

As a rough measure of PEARL's execution speed on the PDP-10. we created a test data base of 4000 structures, in which the average unsuccessful query took 0.0042 CPU seconds (237 per second) and the average successful query took 0.0073 CPU seconds (136 per second). Note that PEARL's hashing mechanism results in fast determination of failure. As another measure of PEARL's execution speed, we compared the original implementation of PAM [5] written purely in LISP with the current implementation using PEARL. The average time required by the original to process a sentence was 5.6 CPU seconds, while the new version requires an average of only 0.56 CPU seconds.

References

[I] Bobrow, D., and Winograd, T. "An Overview of KRL, a Knowledge Representation Language." *Cognitive Scienc§* 1:1 (1977).

[2] Greiner, R., and Lenat, D. "A Representation Language Language." In *Proc. First NCAJ.* Stanford, CA. August. 1980. 165-189.

[3] Roberts, I., and Goldstein, R. "NUDGE. A Knowledge-Based Scheduling Program." In *Proc IJCAI-77.* Cambridge. MA. August. 1977. 257-263.

[4] Schank, R. *Conceptual Information Processing.* Amsterdam: North Holland, 1975.

[5] Wilensky. R. "Understanding Goal-Based Stories", Technical Report 140, Computer Science Department, Yale University, New Haven, CT, September 1978.

[6] Wilensky. R. "Meta-Planning: Representing and Using Knowledge about Planning in Problem Solving and Natural Language Understanding", Berkeley Electronics Research Laboratory Memorandum No. UCB/ERL M80/33. Berkeley. CA. August 1980.

* This research was sponsored in part by the Office of Naval Research under contract N00014-80-C-0732 and the National Science Foundation under grant MCS79-06543.