Visiting ICOT: A Trip report

Catriel Beeri
The Hebrew University

February 14, 1989

Abstract

My visit to ICOT follows an invitation about two years ago by H. Itoh. The purpose of the visit is to learn about the activities in ICOT in certain areas of Data and Knowledge Management, to present the results of my research in these areas, and to discuss the state of the art. The areas of mutual interest are *recursive query processing, nested relation and complex object models,* and *object oriented database models (OODB).* All three areas have been extensively discussed. An important result of the discussions is the emergence of a framework for understanding general recursive query processing methods.

1 Background

Since its inception, ICOT's mandate has been to investigate hardware and software architectures for knowledge management. This is a very wide area. It includes, in particular, recursive query processing over large databases, and advanced models for the representation of both data and knowledge. My work in the years 86 to 87, as represented by papers in PODS 87, involved both recursive query processing, and a development of a recursive language that can handle sets as data elements. The *magic set* method for recursive query processing is quite similar to methods developed in ICOT and related institutions at about the same time, or slightly later. This convergence of ideas led to the extension of the invitation to visit ICOT.

The actual visit was delayed for almost two years. In the meantime, I continued my work on complex object languages, and started to develop ideas concerning object oriented databases. The presentations and discussions during my visit dealt, therefore, with these three subjects.

2 Outline

In this report I summarize the talks I presented, the discussions I had with members of ICOT, and in the last section I present a framework for understanding recursive query processing, that is an important result of these discussions. This last section is technical in nature.

3 Talks

I presented two talks at ICOT, attended by members of ICOT's laboratories, and of the working groups. I also presented a talk at Kyoto University (on February 13). The subjects and main ideas of the talks are described briefly below.

- *Languages for Complex Objects*: This talk described extensions of classical languages to a complex object model, that allows unrestricted application of set and tuple constructors. Both a calculus and an algebra were presented. the main points that were emphasized were:

  1. Both languages need to support restructuring of objects. In the calculus this is accomplished by types attached to variables. In the algebra, it is accomplished by explicit restructuring operations, like project, and by a facility that allows recursive use of operations deep inside objects. The similarity of this facility to those found in general functional languages was emphasized.
  2. In contrast to the languages for flat relations, these languages are very powerful. Their power extends beyond first (and even second) order. The need to control the expressiveness of the languages was pointed out, and a way for doing that was presented.

- *An Object Oriented Database Model*: A major difference between "object oriented" and "complex objects" is the addition of object identity. This talk described a major difficulty in the development of a formal model for object oriented databases, namely, how to provide an interpretation. It was shown that the right approach is not to consider the type equations as defining how the domains for the types are to be constructed. Rather, they only specify the type structures. The domains are supplied as part of any given database structure. It was shown that with this approach, a first order calculus for the model exists, and it was briefly discussed how various additional structural features can be incorporated.

- The talk at Kyoto University combined the subjects of the two talks described above. Rather than describing the algebra, the talk emphasized the power of the complex object calculus. Thus, the talk has illustrated two aspects of a problem that is encountered when we try to extend relational languages to more advanced models: The need to control the expressive power of the languages.

4 Discussions

I was given a general presentation of the goals and structure of ICOT, upon arrival, by Iwata-san. The following days, I was presented with talks about the activities of the third laboratory, by Morita-san, and of the fourth laboratory, by Yokota-san, and on the CAL system, by members of the CAL team.

The description of the activities of the third laboratory included a brief description of the PSI machine, of term relations and unification based operations, the knowledge machine of CHI, and PIM. For me, the unification based operations were of special interest, and their understanding supported the development of the general framework for query processing.

The talk by Yokota-san described in details the KAPPA project. The underlying data model is nested relations, but the operations defining the semantics are different from those that I used in my work. The system also contains deductive capabilities. I was impressed with the scope and diversity of the system, which represents a very serious development effort, and a combination of complex objects and deductive capabilities.

Following these presentations, and before and after my own talks, I had several discussions with members of the laboratories. Several of these meetings were devoted to recursive query processing, with the participation of Miyazaki-san, Seki-san and Morita-san. We discussed the relations between the different methods that were proposed for general recursive query evaluation, including the magic set, the BPA-restrictor, and the magic/Alexander template. The result of these discussions is a better understanding of the assumptions used in each method, and an outline of a general framework for explaining the different techniques and their interactions. This general framework is summarized briefly in the next section.

In another meeting, with the above and with Yokota-san, we explored the applications of recent work on complex objects and object oriented models and languages. My views, as put forward in these meetings can be summarized as follows: (1) When defining a new model and language, it is necessary be be very precise, and to investigate the expressive power of the model and the language. Delicate interactions between the features of the model and the language may exist, and at least for some models that have been considered, the languages seem to be too powerful. (2) There is a wide spectrum of applications for combined data and knowledge management systems, and correspondingly there is a spectrum of possible systems types. For applications with a lot of data, the system model should emphasize structural aspects, as these are central to data manipulation. For other application types, a programming language approach, emphasizing behavior, may be more appropriate.

During the visit to Kyoto University, I had disussions with the following: With Yoshikawa-san, about his recent work on object oriented models. I was interested, in particular, in the work on views, and on behavior analysis. Ibaraki-san told me of his recent work on the complexity of join sequence selection, and on recursive query processing for one-sided recursions. Connections to recent work on such recursions were discussed.

5 A Framework for Recursive Query Processing

Many strategies have been proposed for recursive query processing. They may be classified as *special purpose*, i.e., applicable only to special types of programs, and *general pur-*

pose. This discussion concerns general purpose strategies only. Even this class contains numerous strategies, such as *magic sets* (and generalizations), the *Alexander* method, and the *BPA, restrictor* based, method. These methods were believed to apply to recursive queries over databases only. Very recently, they have been generalized to arbitrary logic programs. The framework is an attempt to explain the different optimization ideas, the roles of various assumptions, and how they relate to each other.

A key idea is that there is a lot of similarity between the bottom-up and top-down approaches. Optimization ideas that apply to one of them can work for the other, although sometimes in a different form. The only one for which this observation seemed at least partially false so far was that of information passing, and the recent developments are closing this gap.

The first observation is that semantics of programs are treated somewhat differently in the database and in the logic programming communities. In databases, only *ground* facts are assumed to be given. The semantics of a program is its least model — a collection of *ground* facts. The answer to a query is the set of all *ground* instances of the query atom that are in the least model (i.e. are implied by the program and the given database). In logic programming, an answer is *any instance* (not necessarily ground) of the query atom that is implied by the program. This difference entails different approaches to computation. The database community favors bottom-up computation, in which only ground facts are generated, using *matching* based joins. This implies that if a rule's head contains a variable that does not appear in the body, than the computation will instantiate this variable with all values in the universe, typically a large (or even infinite) set. For that reason (and others), programs and queries are required to be domain independent, and a covering condition that requires each head variable to appear in the body is imposed. In logic programming, top-down computation, using *unification*, is the paradigm, and non-ground facts are easily accommodated. No covering constraints are imposed.

However, it is possible to use a bottom-up computation for general logic programs: given a collection of facts, new facts can be generated by *unification join* operations corresponding to rule bodies, followed by projection on the heads. Subsumption tests are used to eliminate redundancy. Since facts may contain variables, if some program rules are not covered, the number of facts generated will be smaller than that generated by the database approach. The unification join, however, is much more expensive than matching, or even than unification, as used in top-down computations.

Top-down has an advantage that it can compute only facts that are needed to answer a query, whereas bottom-up needs to compute the complete least model, then it computes the answer by a selection. Note however, that top-down can also be inefficient, if we so desire: We can imagine a top-down computation that starts from a general atom corresponding to each predicate, works on all current unit goals in parallel by unifying each of them with each qualifying rule, thus generating new unit goals, and so on. Of course, once we have answers to the unit goals of a rule's body, we have to "join" them to obtain the answer to the head goal. This is not very different from the unification join used in the bottom-up approach. (It can be viewed as a combination of "and" and

"or" parallelism.) Thus, in a naive approach, bottom-up and top-down are similar.

An obvious optimization of the naive top-down approach is not to use a goal if it (or a more general one) has been used previously. (This is essentially the idea of *memoizing*, also called tabulation, or *lemma resolution*.) A similar idea for bottom-up is the *semi-naive* method, also called the differential method. Another optimization is to restrict attention to the predicate of the query goal. This optimization applies, although only to a limited extent, in the bottom-up approach: we can analyze the program to find out which predicates (and rules) can contribute to the query predicate, and compute only their extensions.

However, the real advantage of top-down is in the utilization of information passing. Giving up "and"-parallelism, we evaluate the atoms in a rule's body is some partial order. (Essentially all compilers actually use a total order.) Bindings for variables obtained by the evaluation of some atoms are available for the following atoms in the rule, and thus more restricted subqueries are generated. Further, when the query contains some binding information, the computation starts from the query goal, rather than from a general goal on the query predicate. It is this optimization, that is inherent in essentially all top-down language processors, that has been hard to imitate in bottom-up computations.

The solution is called by different names: *restrictor, BPA, magic set, Alexander* methods . (These are similar, but non-identical, variations.) In the most general form, developed recently, it works as follows. We can restrict the bottom-up computation by adding to the body of each rule an additional (restrictor, or magic) predicate. This certainly reduces the number of facts generated in the computation, but it may also eliminate some legal answers, depending on the rules defining the new predicates. It turns out that a good choice of such rules is that they compute for the new predicate added to a rule the set of bindings in queries on the head of the rule that can be generated in an optimized computation. On one hand, this choice guarantees that the modified rule will still produce the answers to all queries on its head, and on the other hand, the body will generate only answers to relevant queries, and not a large, partially redundant, set of facts. The new predicate added to a rule has therefore the same arity and same arguments as the rule's head. It is possible to use restrictor predicates with smaller arity; with a good choice of the arguments, there is no loss of efficiency. This is the case, for example, in the magic set method.

The information flow when a rule is processed can be represented by a suitable graph, representing the (partial) order in which the body atoms are evaluated. This graph can be used to generate rules for the additional predicate from the original rule. Essentially, the bindings passed into an atom in a rule's body constitute a new query on that atom's predicate. The new rule captures these bindings in the new restrictor predicate. Note that in contrast to the idea of a sip, as defined in the magic set method, the information that is passed is not represented or used in this general version of the method. The bottom-up computation is optimized simply because we have a container for the information that will be passed at run-time. At compile-time, we only use the knowledge about the direction of the information flow.

Adornments have been presented as part of the magic transformation. But now we

see that, to a first approximation, the use of adornments is an independent idea. In the crudest form, we may consider when information is passed to an atom in a rule body, which arguments will be bound in a meaningful way. That is, their bindings will restrict the computation. We may abstract such knowledge as a string of zeros and ones, where zero means 'no restriction', and one means 'meaningful restriction'. Call such a string an *adornment*. It can be viewed as a representation of a class of queries that will be generated for the corresponding predicate. We may decide to select a processing strategy for a rule, namely the partial order of evaluation, based on the adornment for the head. To do that properly, we use an adornment algorithm. Essentially, for each rule we create as many copies as there are adornments for its head. We obtain a new program, and of course it makes sense to apply a restrictor transformation only after the adornment stage, so that we have a restrictor predicate for each adorned predicate.

An inherent disadvantage of the general restrictor transformation is that in many cases the new rules will not be covered, that is, their head may contains variables not appearing in the body, even if in the original program all rules are covered. Consider, as an example, the well known ancestor example.

$$anc(X,Y) : -par(X,Y)$$
$$anc(X,Y) : -par(X,Z), anc(Z,Y)$$

For a query $anc(john,Y)$, we can use the restrictor predicate $r\_anc(X,Y)$, and the new program is

$$anc(x,Y) : -r\_anc(X,Y), par(X,Y)$$
$$anc(X,Y) : -r\_anc(X,Y), par(X,Z), anc(Z,Y)$$
$$r\_anc(john,Y).$$
$$r\_anc(Z,Y) : -r\_anc(X,Y), par(X,Z)$$

Now, we can see that the second argument of the fact given for r_anc is a variable, and it will contain variables during query evaluation. Thus, in a bottom-up database style computation we need to instantiate it to all elements of the universe.

The first step of the solution is to remove from the restrictor predicates all the positions that do not carry useful restrictions at run-time. Adornment is useful for this optimization, since it separates different information passing patterns from each other. For the example above, we remove the second position of r_anc, and for this example this is sufficient — all rules are now covered. In general, when function symbols may be used, this optimization is insufficient. If we know that a possible binding passed into an atom $p(X,Y)$ in some body is that $X = Y$, then we still need the two positions, Similarly, if the binding instantiates some variables in a term to constants, but leaves some other variables free, than we need this position. In such cases, we must use unification-join in the bottom-up computation, or alternatively instantiate uncovered variables to all possible values. Both alternatives are unattractive.

The solution for the database world is the following: We know that facts in the database are ground. All original rules are covered. It follows that bottom-up database style computation for a derived predicate produces only ground answers, hence the information passed to an atom is always of the form 'variable bound to constant'. We eliminate the positions of the restrictor (magic) predicate in which some variables (appearing in a term in that position) may not be bound at run-time to constants. We are left with positions that at run-time are always completely bound to constants, and it can be shown that the rules of the resulting program are covered, and bottom-up matching based computation can be used. Note that since some positions are omitted, we have a partial restrictor rather than a full restrictor. In particular, we may have eliminated some positions that carry useful binding information. Therefore, the scope of this variant is restricted, compared to the general case.

One small problem: In different rules, the patterns of information passing into atoms of the same predicates may be different. If we eliminate positions according to all these patterns, we may be left with no positions, and no restrictor. The answer here is to use adornments, so we have a separate restrictor predicate for each pattern. Further, we now use a different abstraction for the adornment where zero, or free, means that we cannot guarantee that this argument will be completely bound, and one or bound means we can make this guarantee. The details are as in the magic set paper.

The above outline is necessarily brief. However, it was accepted positively by ICOT researchers with whom it was discussed. This approach leads to some obvious problems for further research: What are the most general cases when matching based, bottom-up computation can be used? Can they be found by efficient algorithms? What is the precise role of adornments? What is the cost of bottom-up unification -join computation, compared to other methods? Finally, can some special purpose techniques also be described in this framework?

6 Acknowledgements

The atmosphere at ICOT has been extremely friendly from the first day. I would like to thank especially Iwata-san for taking care of the initial arrangements, for helping me to rent an apartment (which enabled me to live a little more like a Japanese, rather than like a tourist), and in particular for caring so much about my specific dietary requirements. Thanks are also due to to Morita-san, who arranged my meetings, and took care of many small details, such as instructing me on how to obtain lunch, and so on. Many other members of ICOT did their best to make my stay enjoyable. Finally, thanks to all the people I met for technical discussions, that have contributed to, and helped clarifying the outline and many details of the general framework.

1. Catriel Beeri - short cv

B.Sc. (Math, Physics), M.Sc. (Math) and Ph.D. (Math & CS) from the Hebrew University of Jerusalem. Ph.D. awarded in 1975.

Post-doc fellow in Computer Science Department, University of Toronto (1975/6); Visiting instructor at Department of EECS, Princeton University (1976/7).

Lecturer, Department of Computer Science, Hebrew University, (1977); promoted to senior lecturer with tenure (1978), and to associate professor (1982). Chairman of the department 1983-1985.

Visiting scientist, IBM Research Laboratory, San Jose, summer 1980; visiting professor, School of Applied Science, Harvard University (1982/3); visiting scientist, Computer Corporation of America, Summer 1983; visiting scientist, Microelectronics and Computer Technology Corp (MCC), summer 1986, spring/summer 1987.

Area of Interest: Database and Knowledge Management Systems, including: theory of data models and languages, logic based declarative languages, concurrency control and recovery with emphasis on nested transactions, intelligent DBMSs.

2. Catriel Beeri - academic resume.

My main research activities over the past ten years have been in the theory of database management systems. Although I am in principle interested in all aspects of database systems, I have mainly been involved with the following topics.

Starting in 1975, until about 1984, I have been active in the area of design theory for relational databases. This included the investigation of data dependencies and algorithms for their manipulation, and later research on schema design methods. Among the contributions of this period are axiomatizations and basic algorithms for functional and multivalued dependencies, the development of general types of dependencies and a theory that explains the roles of different types of dependencies in the schema design process.

Since 1982, I have been working on the development of a model and of proof techniques for nested transaction systems. Although transaction management seem to be well understood, nested transactions are still a challenge. They are becoming important now with the advent of object oriented programming systems. I have fdeveloped with colleagues a model and a substantial body of proof techniques, for both concurrency and recovery. Only a small part of these results have been published, I am now preparing additional papers on the subject.

Since 1986, I have been working on the integration of declarative programming methods from the database and the logic programming areas. While the logic programming languages provide the additional expressive power, the database approach emphasizes efficient processing of large amounts of data. My interest so far has been mainly in the development of general methods for optimizing recursive programs for bottom up computation. Another direction is the extension of such languages to data models that allow

richer data structures, notably sets and tuples together.

Another recent development in the database area is the emphasis on data models that are more expressive than the relational model. I have investigated languages for one extension of the relational model - the complex object model. The results characterize the power of versions of algebra, the calculus, and of recursive languages. These results have recently been extended to an object model that allows, in addition to tuple and set constructs, also object identity and ISA relations.

In summary, I see database systems evolving towards richer data models and more expressive, declarative, languages. Such systems will manipulate both data and knowledge. The expressiveness of a model and of the languages defined for it are intimately related. Understanding this relationship and exploring it for the development of more powerful and flexible data and knowledge management systems is a goal that I intend to pursue for the next few years