ICOT Trip Report

Jonas Barklund

UPMAIL, Computing Science Dept.
Uppsala University
P. O. Box 520
S-751 20 Uppsala, Sweden
+46-18-18 10 50

This is an attempt to summarize my visit to ICOT, lasting from June 15th until July 31th, 1987. The immediate cause of my visit was a reward from 'Anders Walls Stiftelse' in Sweden, to be used for international studies. Choosing my destination was easy: UPMAIL and ICOT have had a good relationship in the past and I had met several of ICOT's researchers before and knew their competence in the field. Also I had the opportunity to visit Japan in November 1985 and I found that I liked the country very much.

After a short correspondence over airmail and electronic mail I was pleased to know that my wish to visit ICOT had been granted by Dr. Fuchi. I was warned that the climate of Japan can be somewhat uncomfortable during the summer, but it was the most convenient time to be away from the university, and I could combine it with attending the 4th International Conference on Logic Programming in Melbourne. Unfortunately, the warnings about the weather were quite accurate. The combination of heat and humidity was not something my body had previously experienced for an extended period of time. This will be one of my few unpleasant memories of Japan (the other two being natto and trying to get cash from banks).

Since my own research has been focused at efficient execution of logic programs, I had been assigned to the fourth laboratory. There Miyazaki-san and Sugino-san had been selected to take care of me during my stay (they did, and so did everyone else). They presented me with an agenda for the first two weeks which was filled with presentations by ICOT researchers from the first and fourth laboratories.

1. Presentations Given by ICOT Researchers

I was accompanied during the first half of my stay by Ian Foster from Imperial College, and most presentations were given to both of us.

Iwata: Introduction to ICOT

Iwata-san gave a brief overview of the organization of ICOT. This helped me put the work at the fourth laboratory in its proper context.

Taki: Multi-PSI

Taki-san presented the architecture of the Multi-PSI parallel machine, describing its relations to the PSI-II and PIM projects.

Ishibashi: SIMPOS

Ichibashi-san showed us SIMPOS running on a PSI machine, explained its basic features and let us play with it for a while. It was robust enough to resist our deliberate attempts to break it down.

Ueda: Guarded Horn Clauses

This was not so much of a presentation, since we were all quite familiar with the GHC language, but rather developed into a long but very interesting discussion about future developments of the language, its implementation and usage.

Ichiyoshi: KL1

I must admit that I missed a large part of this presentation due to late arrival (never trust an alarm clock) but we nevertheless had a good discussion about how to execute KL1 in a distributed environment, concerning throwing of goals, sharing of references between processors and more.

Goto: Parallel Inference Machine

Goto-san explained the concept of a PIM and explained many of the problems with choosing a suitable memory management. We were shown results of many simulations and we discussed which conclusions to draw from them.

Kimura, Chikayama: Multiple Reference Bit

Kimura-san described the novel idea of a scheme for truly incremental garbage collection, developed by Chikyama-san and himself. He explained the problems and costs of maintaining the necessary runtime information and what could be gained. Empirical evidence was given by simulations.

Chikayama, Kimura: KL1B

This was a discussion of the abstract machine KL1B for KL1. Since mr. Foster has done similar work on a machine for Flat Parlog (which he reported later in a presentation) it was very stimulating. One important sub-topic was that of implementation of metacall/sho-en and how to detect termination and deadlock.

Nakajima: PSI-I & PSI-II

Nakajima-san gave a very nice presentation of the architecture of the two sequential computers PSI-I and PSI-II and motivations for design decisions. In particular I found his introduction to micro-coding the PSI-II very enlightening.

Sato: PIMOS

Sato-san gave us a report on the current state of the PIMOS operating system for the PIM. Obviously much work remains to be done since it is not yet clear in which language it is to be written, but I found the approach of writing a parallel operating system purely in committed choice logic promising.

2. My work at ICOT

My time at ICOT was spent on two types of work. During the first weeks I tried to learn as much as possible about the work in progress at ICOT, particularly at the fourth laboratory and also discuss my own research with ICOT researchers. Because of the willingness of ICOT researchers to share information with me and spending their valuable time on presentations and discussions, I think I succeeded in this quest.

Because the duration of my visit was comparatively long, it was possible for me to actively contribute some work to the ICOT project. The second half of my stay I worked with a compiler for KL1 in KL1, based on the compiler in Prolog by Kimura-san and Chikayama-san. I will describe this work further below.

I gave three presentations at ICOT during my visit, one about the work at UPMAIL on data parallelism in logic programs and two progress reports on the KL1 compiler. It seems that the Connection Machine™ (which is the most well-known machine for massive data-parallelism) has received much attention in Japan and many researchers have opinions about it. Therefore I could get some feedback on our work although research in this field is not pursued at ICOT.

Our approach to data parallel execution of logic programs is to recognize that the major source of data parallelism in a resolution-based system is in unification of large terms. In Prolog large terms are rarely unified and the amount of parallelism in unification is too small to be significant. Instead we look for another inference system than SLD-resolution which allows us to quickly build up large terms and then do the unification. The advantage of such a logic programming language compared to other suggested (functional or imperative) languages for data parallel computers is that it seems possible to get massive parallelism (for certain algorithms achieving execution times proportional to the logarithm of the size of input data) without having to explicitly write the parallelism into the program.

3. KL1C compiler in KL1C and programming in KL1C

Before my visit I had never written a big program in a committed-choice language such as KL1. Therefore I found it a great opportunity to get a chance to study the advantages and disadvantages of programming in these languages as well as to understand KL1 better by study its compilation.

The environment which seemed most suitable at the time for developing the compiler was Ueda-san's GHC system on the DEC-20 computer. It compiles Flat GHC to Prolog and executes it with a runtime system which is also written in Prolog. It is written in Prolog-10 (Edinburgh Prolog for the DEC-10 computer) and can not run very large programs because of memory limitations. My first task was therefore to rewrite the GHC system to run on Tricia Prolog which is a native Prolog system developed at UPMAIL for the DEC-20. Since it is largely compatible with Edinburgh Prolog this was not very difficult.

When I started to program in the system I introduced some new constructs in the language and also came up with some programming techniques and constructions which may not be all brand new but still worth reporting. I will introduce them one by one.

Definite Clause Grammar

I introduced into the GHC system as well as the KL1 compiler itself a DCG facility analogous to the one in Prolog. It expands a DCG clause of the form

```
H :- G | B
```

to a GHC clause

```
H' :- G | B'
```

where in `H'` and `B'` all non-terminals have been expanded with two extra variable arguments, sharing variables between adjacent non-terminals. This extension was quite straightforward to implement (not trivial though, since the suspension rules of GHC may play games with you if you are not careful) but very useful for the multiplexed stream programming described below.

Case construction

In Prolog disjunctions are sometimes useful in order to obtain alternative execution without having to introduce a predicate which is called only once. Except for the behaviour of cut, a disjunction can be immediately replaced by a call to a new predicate defined by as many clauses as there are alternatives in the disjunction. The analogous construction in KL1/GHC would be something similar to the if-then-else construction of Prolog, where every alternative of a disjunction is required to have a guard. I implemented this in the GHC system and in the KL1 compiler. A disjunction

```
G1 -> B1 ;
G2 -> B2 ;
...
Gn -> Bn
```

occurring in the body of a clause is replaced by a call to a new predicate (whose name is cooked up to be unique enough) and whose arguments are all variables in the clause occurring both inside and outside of the disjunction. (An earlier approximation which as arguments took all variables occurring in the disjunction did not work because it would cause undesirable infinite suspensions on void variables.) The new predicate is defined by `n` clauses, where the guard and body of the `i`th clause is `Gi` and `Bi` with variables renamed.

Multiplexed streams

The DCG facility is very useful when a predicate is expected to return a stream. Items can be placed in the stream simply by making them DCG terminals. However, one does often want to output several streams from a predicate. It is difficult to invent a convenient syntax for extending the DCG facility to clauses constructing several streams and introducing two extra arguments per stream can also be quite costly. The solution I chose (inspired by a discussion with Ian Foster) is to use only one output stream, tagging each item placed in the stream so that later it can be determined where it really should go. For example, in one part of the compiler we want to send things to be written on the terminal to an I/O-stream, produce a sequence of instructions and also communicate with a perpetual process inventing new source variables. This would require three streams. As it is, everything is placed in only one stream which is sent to a perpetual process having one instream and three outstreams. Each item in the instream is examined and sent to the right outstream. Programming this way has been very conventient. I still do not know the overhead for handling the multiplexing but it appears to be smaller than the overhead for handling several streams instead of one and the programming time has definitely been very much reduced.

Negation of tests

To make two clauses mutually exclusive and independent of the execution mechanism one often wants them to contain tests which are disjunctions of each other. Therefore, in the GHC system and the KL1 compiler, I allowed any primitive test `T` to be negated as `not T`. This was very handy and made it possible to avoid a number of predicates with names such as noninteger, nonatomic etc.

4. KL1

Clause selection and indexing of KL1

I often discussed details of KL1B, particularly with Nakajima-san, Kimura-san and Chikayama-san. It seems that the fact that clause order is not important in KL1 should be used to compile code so that tests are never repeated unnecessarily. The whole guards should be used for indexing, since they often contain useful information about the mutual exclusiveness of clauses. A simple indexing scheme depending only on one or more head arguments will not be the most efficient. It is possible to compile the guards of all clauses together to a tree or an acyclic graph where each internal node is a simple test or switch and each leaf is the body of some clause or suspension. No path would have to contain the same test or switch more than once. I think a possible way of creating the tree is by expressing each variable occurring in the head and guard as a function of the head arguments. This way it is possible to notice equalities between variables and tests and negations of tests in different clauses, to avoid re-evaluating expressions and tests and use mutual exclusivity.

At runtime, this requires some ability to distinguish between failure and suspension when testing for type, arithmetical order etc. The current approach seems to be separating the check for instantiation from the instruction performing the actual test and thus allowing different action when arguments are not sufficiently instantiated. This works in many cases, but not when two arbitrary terms are compared for equality, since they may suspend because of variables occurring deep in the terms. I suggested splitting the register holding the adress where to jump on failure and suspension in two, one for failure and one for suspension. Nakajima-san suggested an alternative, introducing a bit for quick suspension, which seems functionally equivalent with my idea but is probably easier to implement.

Guard predicates

Since guards in KL1C may only contain predefined predicates it is important to choose them carefully. Programming the compiler, I often used predicates for testing types, performing arithmetic, decomposing structures (`functor/3` and `arg/3` of Prolog) and comparison. I also used their negation frequently.

Otherwise

I think it is necessary to have some kind of sequential or for clause selection. With one clause for each of a hundred special cases and then a catch-all clause it is ridiculous to require the user to write a guard for the catch-all clause which makes it exclusive from all the others. Relying on clauses to be tested sequentially is not a solution, since it does the same harm to declarative semantics as cut does in Prolog and destroys some possible compiler optimisations.

On the other hand, such a feature should not be used unnecessarily, a compiler which compiles indexing cleverely, according to above can probably generate better code for a set of clauses with mutually exclusive guards than if they are interspersed with 'otherwise'.

Meta-level programming support

In Prolog variables can be embedded in partially instantiated data structures and a predicate does not have to know which parts of a data structures are instantiated and which are not. This is not true in KL1 where an attempt to inspect a variable no-one else is going to instantiate causes infinite suspension, and attempts to instantiate something which is already instantiated to something else causes failure of the whole system (at least within one sho-en). Some compiler writers using Prolog use this as a feature for writing unintelligible but efficient compilers.

In Ueda-san's GHC system this is cured by replacing all variables in terms which are input from terminals or files by `'$VAR(`X`)'` where X is some integer unique to that term. This works in many cases but falls down completely when compiling a program containing explicit occurrences of `'$VAR(...)'`, such as the

KL1 compiler itself. Therefore it is necessary to think of some more advanced mechanism for distinguishing terms (or at least veriables) at different meta-levels.

Arrays

It is suggested that mutable arrays should be incorporated in KL1 and I support this idea, since updating of arrays can often be made very efficient using the MRB scheme. I introduced arrays in the GHC system, which was possible since the underlying Tricia system has them. I often found it useful to access array elements from guards, which should be allowed since it is just another way of decomposing a structure.

Making guards stronger

Sometimes the restriction that guards may not contain user-defined goals is very annoying. It was often the case when writing the compiler that I wanted to use one clause if a certain argument had one of a number of values and another clause for some other values etc. The alternatives were to either have one almost identical clause for each possible value or to make the guard empty and in the body first call a predicate returning different things for each group of allowed values and next call a new predicate which dispatches on this thing. I usually chose the second approach and then found the case construction useful to avoid introducing auxilliary predicates.

Debugging

Somewhat to my surprise I found that when a deadlock occurred there were usually only few goals left, typically 5 to 15 for my program. However, they sometimes contained very large data structures. A pretty-printer able to limit the depth to which subterms are written and the length of lists written would enable many deadlocked goal systems to fit on a normal computer screen. This would also be useful when tracing the execution of a program.

When a deadlock has occurred the system should upon request tell which variables have only one occurrence in the system and in which clause they originated. Restricting this to be only those void variables upon which goals are actually suspended would be even better.

Failure of a goal in a system normally means failure of the whole system. Using a sho-en it is possible to trap the failure and do domething about it. However, during debugging of a program it is often the case that goals fail because of program bugs. When this happens, the user should have an option to proceed normally with the failure as above, or to ignore the failed goal and continue execution, perhaps given bindings of some output variables in the failed goal to prevent infinite suspension on its output. The reason for this is that one wants to find as many bugs as possible without having to edit and recompile the program and if the program terminates with failure, at most one such bug can be detected.

KL1nt Eastwood

One strength of the C programming language is the existence of a program which statically checks many aspects of a program, detecting many bugs before the program is even run. This would be very useful for KL1, since many infinite suspensions (e.g., void variables) and failures (e.g., non-existent predicates, type mismatches) could be detected this way.

5. My Impression of ICOT Research

On the whole it seems to me that the Fourth laboratory is reasonably on schedule. I am told that the current goal is to have the multi-PSI running at the Fifth Generation Computer Systems conference in November 1988. I think that there is a fair chance to succeed in this but it is going to take a lot of hard work.

The architecture of the multi-PSI but also of PIM is more conventional than one might have expected from the initial goals set up for ICOT. Dr. Uchida gave an explanation for this; if both the software people and the hardware people had gone for something new and unconventional, neither of them would have had any solid ground on which to stand and very much time would have been lost. I think this is a good observation, but I hope the quest for new efficient and parallel computer architectures has not ended by this.

One thing I find somewhat disappointing is that the tight 10-year schedule of ICOT gives the developers of programming languages and environments too little time to evaluate their work. The machine on which KL2 is going to run is already being designed even though the work on KL2 has not started seriously. Not

even KL1 is completely fixed yet and there is only little experience of using it. I hope this is not going to harm the construction of KL2 and its efficient execution, but it is still too early to tell.

Another concern of mine is where the logic programming has gone. The committed choice language family, of which GHC and KL1 are members, lacks some of the features I would associate with a logic programming language. The proposed KL1U language intends to support object-oriented programming. I am told that in the beginning it was believed that this should be just a nice syntax and that the programs could be mechanically transformed to KL1C programs. The current opinion seems to be that this is not necessary and there is work on how to execute KL1U directly. There need not be anything wrong in this but I think it is important to pursue *one* well-defined line of research. So far this line has been along the lines of logic programming. I think diversifying the research will only make ICOT weaker and I would not want this to happen.

As has been noticed at ICOT and many other research centers, there is a lack of applications created in new languages. I suppose that the problem is that the people constructing languages are too busy to write any other application than perhaps a compiler in their new language. Applications writers usually can not take the risk of using a newly implemented language with little support and simple environment when they write a serious application, so instead they go for languages which are already used. Prolog is starting to overcome this, since it is nowadays a fairly well-known language with several serious, rich and stable commercial implementations. Languages such as KL1 will not reach this stage for a long time, but the language designers would surely need the feedback. I am afraid I can not suggest any safe solution to this, except continued work to make implementations which are of sufficiently high standard and with some guaranteed future maintenance so application writers find them competitive.

However, I am optimistic about the outcome of the second half of ICOT's term. There are plenty of excellent researchers at ICOT and they work very hard. I do not think they will disappoint us who are watching.

6. Acknowledgments

I am very thankful to all those who made my visit possible and so enjoyable. My first thanks go to Dr. Fuchi and Dr. Uchida for allowing me to stay at ICOT's Fourth laboratory for seven weeks in a very busy period. Secondly I would like to thank the individual researchers who were kind enough to take very good care of me. I spent much time with Ichiyoshi-san who is a good friend I very much look forward to see again. He is learning some Swedish and I hope to host him here sometime. Rokusawa-san is another memorable person who among other things took us on a nice hiking expedition to the top of a mountain in the Tokyo area. Yoshida-san is an exceptional Japanese lady who has been a great companion and with whom I have had many interesting discussions. Kimura-san was a valuable working companion, it is thanks to his compiler in Prolog that my work on a compiler in KL1 was so easy. Chikayama-san is a very important person at the Fourth laboratory with many good ideas, discussions with him were very stimulating. My hosts, Miyazaki-san and Sugino-san, did a lot to make my stay trouble-free. Iwata-san and Nakamura-san helped me with tickets etc, and were very patient with my sometimes changing plans. I enjoyed the visit to Iwata-san's home very much. Also I must mention Nakajima-san, Suzaki-san and Taki-san who were all very kind to me. It was a great pleasure for me to treat Ichiyoshi-san, Rokusawa-san, Suzaki-san and Yoshida-san at Swedish restaurants, to return some of their hospitality and I greatly appreciate their interest in Swedish culture. Okumura-san and Sakama-san of the first laboratory often closed ICOT together with me late nights. Dr. Futamura at Hitachi Advanced Research Laboratories arranged my accomodation very well at a walking distance from ICOT, I am very grateful to him and his mother for this. Ian Foster is a good fellow, we had both good discussions and fun. I could not have made the visit without the reward from 'Anders Walls Stiftelse' and 'Upplands nation' in Uppsala. I also owe much to Anna, the woman I live with, and Ellin, my daughter.

One of my most pleasant memories is Japanese meals. They are composed of good food and good company. I write this back in Sweden and I am already looking forward to having them both again.