Kernel Language in the Fifth Generation Computer Project

Kazunori Ueda
Computer Systems Research Laboratory
NEC C&C Systems Research Laboratories
1-1, Miyazaki 4-chome, Miyamae-ku, Kawasaki 213, Japan
Tel: +81-44-856-2178 FAX: +81-44-856-2231
`ueda@csl.cl.nec.co.jp`

1 Introduction

This article reviews the design process of KL1, ICOT's kernel language for the Parallel Inference Machine.

An outstanding feature of the Fifth Generation Computer Project is its middle-out approach. Logic programming was chosen as the central notion with which to link highly parallel hardware and application software, and three versions of so-called *kernel language* were planned, all of which were assumed to be based on logic programming. The three versions corresponded to the three-stage structure of the project: initial, intermediate, and final stages.

The first kernel language, KL0, was based on Prolog and designed in 1982 as the machine language of the Sequential Inference Machine. Initial study of the second kernel language, KL1, for the Parallel Inference Machine started in 1982 as well. The main purpose of KL1 was to support parallel computation. The third kernel language, KL2, was planned to address high level knowledge information processing as well. Although ICOT conducted research on languages for knowledge information processing throughout the project and finally proposed the language Quixote [1], it was not called a "kernel" language which meant a language in which to write everything. Of these languages, this article will focus on the design and the evolution of KL1, in which I was involved for years since 1983.

What are the implications of the middle-out approach to language design? In a bottom-up or top-down approach, language design could be justified by external criteria such as amenability to efficient implementation on parallel hardware and expressive power for knowledge information processing. In the middle-out approach, however, language design must have strong justifications in its own right.

The design of KL0 could be based on Prolog which was quite stable when the FGCS project started. In contrast, design of KL1 had to start with finding a counterpart of Prolog, namely a stable parallel programming language based on logic programming.

Such a language was supposed to provide a common platform for people working on parallel computer architecture, people working on parallel programming and applications, people working on foundations, and so on.

It is well-known that ICOT chose logic programming as its central principle, but it is less well-known that the shift to concurrent logic programming started very early in the research and development of KL1. Many discussions were made during the shift, and many criticisms and frustrations arose even inside ICOT.

In these struggles, I proposed Guarded Horn Clauses (GHC) as the basis of KL1 in December 1984. GHC was recognized as a stable platform with a number of justifications, and the basic design of KL1 started to converge. Thus it should be meaningful to present how the research and development of KL1 was conducted and what happened inside ICOT before KL1 became stable in 1987. The article also presents my personal principles behind the language design and perspectives on the future of GHC and KL1.

2 Joining the FGCS Project

When ICOT started in 1982, I was a graduate student of the University of Tokyo. My general interest at that time was in programming languages and text processing, and I was spending most of the time on the thorough examination of the Ada reference manual as a member of the Tokyo Study Group of Ada (Takashi Chikayama was also a member of the Tokyo Study Group). A colleague Hideyuki Nakashima, one of the earliest proponents of Prolog in Japan, was designing Prolog/KR [13]. We and Satoru Tomura started to write a joint paper on input and output (without side effects) and string manipulation facilities in sequential Prolog, with a view to using Prolog as a text processing language instead of languages like SNOBOL.

The work was partly motivated by our concern about declarative languages: We had been very concerned about the gap between the clean, "pure" version of a language for theoretical study and its "impure" version for practical use. I was wondering if we could design a clean, practical and efficient declarative language.

On concurrent programming, I had been disappointed with language constructs for concurrency because of their complexity. However, Hoare's enlightening paper on CSP (Communicating Sequential Processes) [11] convinced me that concurrent programming could be much simpler than I had thought.

I joined NEC Corporation in April 1983 and soon started to work on the FGCS project (ICOT had no position for a new researcher in 1983, but NEC had a commission program to work with ICOT, as did other Japanese computer manufacturers). I was very interested in joining the project because it was going to design new programming languages, called kernel languages, for knowledge information processing (KIP). The kernel languages were assumed to be based on logic programming.

It was not clear whether logic programming could be a base of the kernel language that could support the mapping of KIP to parallel computer architecture. However, it seemed worthwhile and challenging to explore the potential of logic programming in that direction.

3 KL1 Design Task Group

3.1 Prehistory

The study of KL1 had already been started when I joined the project.

ICOT research on early days was conducted according to the 'scenario' of the FGCS project established before the commencement of ICOT. Basic research on the Parallel Inference Machine (PIM) and its kernel language, KL1, started in 1982 in parallel with the development of the Sequential Inference Machine and its kernel language, KL0. Koichi Furukawa's laboratory was responsible for the research into KL1.

Although KL1 was supposed to be a machine language for PIM, the research into KL1 was initially concerned with higher-level issues, namely expressive power for the description of parallel knowledge information processing (e.g., knowledge representation, knowledge-base management and cooperative problem solving). The key requirements to KL1 included the description of a parallel operating system as well, but this again had to be considered from higher-level features (such as concurrency) downwards, because ad-hoc extension of OR-parallel Prolog with low-level primitives was clearly inappropriate. The project was concerned also with how to reconcile logic programming and object-oriented programming which was rapidly gaining popularity in Japan.

Research into PIM, at this stage, focused on parallel execution of Prolog. Concurrent logic programming was not yet a viable alternative to Prolog, though, as an initial study, Akikazu Takeuchi was implementing Relational Language in Maclisp in 1982. It was the first language which exclusively used guarded clauses, namely clauses with guards in the sense of Dijkstra's guarded commands. Ehud Shapiro proposed Concurrent Prolog that year, which was a more flexible alternative to Relational Language that featured read-only unification. He visited ICOT in October–November 1982 and worked on the language and programming aspects of Concurrent Prolog mainly with Takeuchi. They jointly wrote a paper on object-oriented programming in Concurrent Prolog [16]. The visit clearly influenced Furukawa's commitment to Concurrent Prolog as the basis of KL1.

3.2 The Task Group

After I joined the project in April 1983, I knew that the project was aiming at much more general-purpose computing than I had expected. Koichi Furukawa was always saying that what we were going to build was a "truly" general-purpose computer for the 1990's. He meant that the emphasis must be on symbolic (rather than numeric) computation, knowledge (rather than data) processing, and parallel (rather than sequential) architecture.

As an ICOT activity, the KL1 Design Task Group started in May 1983.[1] Members included Koichi Furukawa, Susumu Kunifuji, Akikazu Takeuchi and myself. The deadline of the initial proposal was set to August 1983 and intensive discussions started.

[1] Very fortunately, I found a number of old files of the Task Group in ICOT's cellar, which enabled me to present the precise record of the design process here.

By the time the Task Group started, Furukawa and Takeuchi were quite confident of the following guidelines:

1. (Concurrent Prolog) The core part of KL1 should be based on Concurrent Prolog, but should support search problems and meta-programming as well.
2. (Set/stream interface) KL1 should have a set of language constructs that allows a Concurrent Prolog program to handle sets of solutions from a Prolog engine and/or a database engine and to convert them to streams.
3. (Meta-programming) KL1 should have meta-programming features that support the creation and the (controlled) execution of program codes.

Apparently, set/stream interface was inspired by Clark *et al.*'s work on IC-PROLOG [2], and meta-programming was inspired by Bowen's and Kowalski's work on meta-programming [2]. The idea of sets as first-class objects may possibly have been inspired by the functional language KRC[18].

I knew little about Relational Language and Concurrent Prolog until I joined the project. I was rather surprised by their bold decision to abandon Prolog's feature to search solutions, but soon accepted the decision and liked the language design because of the simplicity.

Various issues related to the above three guidelines were discussed in nine meetings and a three-day workshop, until we finally agreed upon those guidelines and finished the initial proposal.

We assumed that KL1 predicates (or relations) be divided into two categories, namely *AND relations* for stream-AND-parallel execution of concurrent processes based on don't-care nondeterminism, and *OR relations* for OR-parallel search based on don't-know nondeterminism. The clear separation of AND and OR relations reflected the fact that the OR relations were assumed to be supported by a separate OR-parallel Prolog machine and/or a knowledge-base machine. Years later, however, we decided not to create machines other than PIM but to support search and database applications by software, when we became confident that it could be done with reasonable performance. Set/stream interface was to connect these two worlds of computation. We discussed various possible operations on sets as first-class objects.

Meta-programming was being considered as a framework for

- the observation and control of stream-AND-parallel computation by stream-AND-parallel computation, and
- the observation and control of OR-parallel computation by stream-AND-parallel computation.

The former aspect was closely related to operating systems and the latter aspect was closely related to the set/stream interface. Meta-programming was supposed to provide a protection mechanism also. The management of program codes and databases was another important concern. Starting with the 'demo' predicate of Bowen and Kowalski, we were considering various execution strategies and the representation of programs to be provided to 'demo'.

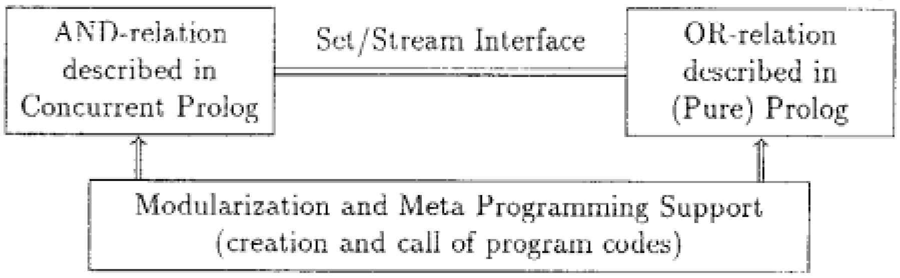
AND-relation described in Concurrent Prolog
Set/Stream Interface
OR-relation described in (Pure) Prolog
Modularization and Meta Programming Support (creation and call of program codes)

Figure 1: Conceptual Configuration of KL1 (1984) [7]

Other aspects of KL1 considered in the Task Group included data types and object-oriented programming. It was argued that program codes and character strings must be supported as built-in data types.

The initial report, "Conceptual Specification of the Fifth Generation Kernel Language Version 1 (KL1)" was published as an ICOT Technical Memorandum in September 1983, which comprised six sections:

1. Introduction
2. Parallelism
3. Set Abstraction
4. Meta Inference
5. String Manipulation
6. Module Structures

In retrospect, the report presented many good ideas and surprisingly well covered the features that were realized in some form or other by the end of the project, though of course, the considerations were immature. The report did not yet consider how to integrate those features in a coherent setting. The report did not yet clearly distinguish between features requiring hardware support and those realizable by software.

ICOT invited Ehud Shapiro, Keith Clark and Steve Gregory in October 1983 to discuss and improve our proposal. Clark and Gregory had proposed the successor of the Relational Language, PARLOG [5]. A lot of meetings were held (when ICOT researchers had difficulties in discussing in English to a greater or lesser degree), and many ICOT people outside the Task Group attended as well.

In the discussions, Shapiro criticized the report as having too many good ideas, and insisted that the kernel language should be as simple as possible. He tried to show how a small number of Concurrent Prolog primitives could express a variety of useful notions including meta programming. While Shapiro was exploring a meta-interpreter approach to meta-programming, Clark and Gregory were pursuing a more practical approach in PARLOG, which used the built-in 'metacall' primitive with various features.

From the implementation point of view, most of us thought that the guard mechanism and the synchronization primitive of PARLOG were easier to implement than those of Concurrent Prolog. However, the KL1 Design Task Group stuck to Concurrent

Prolog for the basis of KL1; PARLOG as of 1983 had a lot more features than Concurrent Prolog and seemed less appropriate for the starting point. Some people were afraid that PARLOG imposed too static dataflow concepts, making programming less flexible.

The discussions with the three invited researchers were enlightening. The most important feedback, I think, was that they reminded us of the scope of KL1 as the *kernel* language and lead us to establish the following principles:

- Amenability to efficient implementation,
- Minimal number of primitive constructs (cf. Occam's razor),
- Logical interpretation or program execution.

Meanwhile, Furukawa started to design a user-level language on top of KL1. The language was first called Popper (Parallel Object-oriented Prolog Programming EnviRonment), and then Mandala. On the other hand, the implementation aspect of KL1 was left behind for a long time, until Shapiro started discussions of sequential, but serious, implementation of Concurrent Prolog. The only implementation of Concurrent Prolog available was an interpreter on top of Prolog, which was not fast –a few hundred RPS (reductions per second) on DECsystem-20.

After the three invited researchers left, the Task Group had many discussions on the language specification of KL1 and the sequential implementation of Concurrent Prolog. Although we started to notice that the informal specification of Concurrent Prolog left some aspects (including the semantics of read only unification) not fully specified, we became convinced that Concurrent Prolog was basically the right choice, and started to convince the ICOT members and the members of relevant Working Groups from January 1984.

Three large meetings on Concurrent Prolog were held in February 1984, which many people working on the FGCS project attended. The Task Group argued for Concurrent Prolog (or concurrent logic programming in general) as the basis of KL1 on the following grounds:

1. It is a general-purpose language in which concurrent algorithms can be described.
2. It has added only two syntactic constructs to the logic programming framework.
3. It retains the soundness property of the logic programming framework.
4. Elegant approaches to programming environments taken in logic programming could be adapted to concurrent logic programs.

People gave implicit consent to the choice of the Task Group in the sense that nobody proposed an alternative basis of KL1 in response to our solicitation. However, as a matter of fact, people were quite uneasy about adopting Concurrent Prolog as the basis of KL1. The arguments being made by the Task Group looked like belief rather than evidence. Many people, particularly those working on PIM, were rather upset (and possibly offended) that don't-know nondeterminism of Prolog was going to be excluded from the core part of KL1 and moved to a back-end Prolog engine. Unlike in logic programming, the direction of computation was more or less fixed, which was considered inflexible and unduly restrictive. However, Furukawa maintained that the parallel symbolic processing was a more important aspect of KL1 than exhaustive search.

Implementation people had another concern: whether reasonable performance could be obtained. Some of them even expressed that it could be too dangerous to have parallel processing as the main objective of the FGCS project.

Anyway, through the series of meetings, people agreed that a user language must be higher-level than Concurrent Prolog and that various knowledge representation languages should be developed on top of the user language. We agreed also that programming environments for Concurrent Prolog (and a KL1 prototype) must be developed quickly in order to accumulate experiences with concurrent logic programming. We decided to develop a Concurrent Prolog implementation in a general-purpose language (C was considered first; Maclisp was chosen finally) to study implementation techniques.

In March 1984, the Task Group finalized the report on the Conceptual Specification of KL1 and published it as an ICOT Technical Report [7]. The report now concentrated better on the primitive features to be supported directly in KL1 for flexible and efficient KIP.

3.3 Implementing Concurrent Prolog

A good way to understand and examine a language definition is to try to implement it; it forces us to consider every detail of the language. From April 1984, the Task Group had some new members, including Toshihiko Miyazaki, Nobuyuki Ichiyoshi and Jiro Tanaka, and started a project on the sequential implementation of Concurrent Prolog under the coordination of Takeuchi. We decided to build three interpreters in Maclisp, which differed in the multiple environment mechanisms necessary to evaluate the guard parts of program clauses. The principal member, Miyazaki, was so quick in designing and Lisp programming. We also started to build a Mandala implementation in Concurrent Prolog.

As an independent project, Chikayama started to improve on Shapiro's Concurrent Prolog interpreter on top of Prolog. By compiling program clauses to some extent, he improved the performance to 4kRPS, a much better number for practical use. I further improved the performance by compiling clauses to a greater degree, and obtained 11kRPS by November 1984, a number better than most interpretive Prolog systems in those days.

We had a general question on the implementation of concurrent logic languages as well, which had been mentioned repeatedly in our discussions on systems programming. The question was whether we could implement basic operations such as many-to-one interprocess communication and array updates as efficiently as in procedural languages in terms of time complexity. For systems programming without side effects to be practical, it seemed essential to show that the complexity of these operations is not worse than that of procedural languages. I devised an implementation technique of these operations with Chikayama in the beginning of 1984, and presented it in the FGCS'84 conference. These two pieces of work on implementation convinced myself of the viability of concurrent logic programming languages as the basis of KL1.

Meanwhile, Clark visited us again in spring 1984, and introduced a revised version of PARLOG [6]. The language had been greatly simplified. Although we were too committed to Concurrent Prolog at that time, the new version influenced the research on KL1 later in various ways.

The three Concurrent Prolog interpreters were almost complete by August 1984, and an interim report comparing the three methods was written. Two successor projects started in September, one on an abstract KL1 machine architecture, and the other on an KL1 compiler. I started to design an abstract machine instruction set with Miyazaki, but was not very excited about it. One reason is that we had found several unclarified points in the definition of Concurrent Prolog, most of which were related to read-only unification and the execution of the guards of program clauses. I started to feel that we should re-examine the language specification of Concurrent Prolog in depth before we went any further. The other reason is that full implementation of the guard construct seemed to be too complicated to be included in a KL1 implementation. The idea of Flat Concurrent Prolog (FCP), which avoided the nesting of guards by allowing only simple test predicates in guards, was conveyed to us from Shapiro in June 1984, but few of us, including myself, were interested.

In retrospect, it is rather curious that we stuck to the full version of Concurrent Prolog which was hard to implement. However, we were not confident of moving to any subset. The guard construct, if successfully implemented, was supposed to be used for OR-parallel problem solving and for the protected execution of user programs in an operating system.

People working on PIM, who were supposed to implement KL1 in the future, were getting impatient in mid 1984. As architects, they needed a concrete specification of KL1 as early as possible and wanted to know what kinds of operations should be particularly optimized, but the design of KL1 had not reached such a phase. On the other hand, members of the KL1 Design Task Group were unhappy that they received few constructive comments from outside. A kind of mutual disbelief was exposed in three meetings of the PIM Working Group held from June to August, in which the Task Group discussed with the PIM people.

4 Proposal of GHC, a New Basis of KL1

After the FGCS'84 conference in November 1984, I started to re-examine the language specification of Concurrent Prolog in detail, the main concerns being the atomicity (or granularity) of various operations, including read-only unification and commitment, and the semantics of the multiple environment mechanism [19]. Many subtle points and difficulties were found and discussed. I had to conclude that although the language rules could be made rigorous and coherent, the resultant set of rules would be more complex and require more sequentiality than we had expected.

The result of that work was not very encouraging, but anyway, I sought for a coherent set of language rules. In mid December, I came up with an alternative to Concurrent Prolog, which solved the problems with read-only unification and the problems with the multiple environment mechanism simultaneously. The idea was to suspend the computation of the guard of a clause if it would require a multiple environment mechanism, that is, if the computation would instantiate variables in the caller of the clause. The semantics of guard now served as the synchronization mechanism as well, making read-only unification unnecessary.

On December 17, I proposed the new language to the KL1 Design Task Group as KL0.7. The name KL0.7 meant the core part of KL1 that left

- the decision on whether to include pure Prolog to support exhaustive search directly,
- machine-dependent constructs, and
- the set of predefined predicates.

The handout (in Japanese) included the following claims:

1. Read-only annotation is dispensed with because it does not fit well in fine-grain parallel computation models.
2. Multiple environments are unnecessary. It is not yet clear whether multiple environments must be implemented, while it certainly adds to implementation cost. Multiple environments make the visibility rule (of the values of variables) and the commitment rule less clear.
3. Mode declaration is dispensed with; it can be excluded from the kernel language level.
4. One kind of unification is enough at the kernel language level, though a set of specialized unification primitives could exist at a lower level.
5. Implementation will be as easy as PARLOG.
6. Implementation will be as efficient as PARLOG.
7. A meta-interpreter of itself can be written.
8. Sequentiality between primitive operations is minimized, which will lead to high intrinsic parallelism and clearer semantics.

Interestingly, the resultant language turned out to be semantically closer to PARLOG than to Concurrent and FCP in the sense that it was a single-environment language. Unlike PARLOG, however, it did not assume static analysis or compilation; PARLOG assumed compilation into Kernel PARLOG, a language with lower-level primitives. The handout claimed also that pure Prolog need not be included in KL1 if we made sure that exhaustive search could be done efficiently in KL1 without special support.

The only new aspect to be considered in the implementation of KL0.7 was the management of nested guards. I found that it could be done anyway and expected that static analysis would help in many cases. It was not clear whether complex nested guards were really necessary, but they were free of semantical problems and thus could be retained for the time being. In addition, the new language was undoubtedly more amenable to compilation than Concurrent Prolog.

I quickly finished two internal reports, "Concurrent Prolog Re-Examined" and "A Draft Proposal of CPII" and brought them to the Task Group meeting on December 20. The name CPII (Concurrent Prolog II) was selected tentatively and was used for a while. The Task Group seemed to welcome my proposal and appreciated the simplification.

In January 1985, the Task Group restarted to design KL1 with a new basis. Takeuchi proposed that KL1 be CPII with the metacall construct *à la* PARLOG and primitives for the allocation and scheduling of goals. The proposal well reflected the final structure of the core part of KL1. Set/stream interface and modularization (as a user-level feature) were still considered to be part of KL1, but were put aside for the moment.

Anyway, by January 1985, the Task Group reached an agreement to base KL1 on CPII. The agreement was quick and without so many discussions, because we had agreed to base KL1 on some concurrent logic language, and CPII seemed to have solved most of the problems we had felt with Concurrent Prolog. CPII did exclude some of the programming techniques allowed in Concurrent Prolog, as Shapiro's group at the Weizmann Institute pointed out later. However, we preferred a language which was simpler and easier to implement.

People outside the Task Group also welcomed the proposal of CPII, though most of them were not yet convinced of the approach based on concurrent logic programming in general. It was not very clear, even to us in the Task Group, how expressive this conceptual language was in a practical sense, much less how to build large parallel software in it. However, there seemed to be no alternative CPII as long as we were to go with concurrent logic programming, since the language seemed to address "something essential".

In early January, Masahiro Hirata at Tsukuba University, who was independently working on the formal operational semantics of Concurrent Prolog, let me know that the functional language Qute designed by Masahiko Sato and Takafumi Sakurai [15] had incorporated essentially the same synchronization mechanism. The news made me wonder if the essence of CPII was simply the rediscovery of a known idea. After learning that Qute introduced the mechanism to retain the Church-Rosser property in the evaluation of expressions, however, I found it very interesting that the same mechanism was independently introduced in different languages from different motivations. This suggested that the mechanism introduced in these languages was more universal and stable than we had thought at first. Apparently, Hirata was independently considering an alternative to the synchronization mechanism of Concurrent Prolog, and later proposed the language Oc [10], which was essentially CPII without any guard goals.

By January 21, I modified my Concurrent Prolog compiler on top of Prolog and obtained a CPII compiler. The modification took less than two days, and demonstrated the suitability of Prolog for rapid prototyping. Miyazaki also made a GHC compiler with more features by modifying Chikayama's Concurrent Prolog compiler on top of Prolog.

In the meantime, I considered the name of the language by putting down a number of keywords on my notebook. The name was changed to Guarded Horn Clauses (GHC) by February 1985.

In March 1985, the project on Multi-SIM (renamed to Multi-PSI later) started under

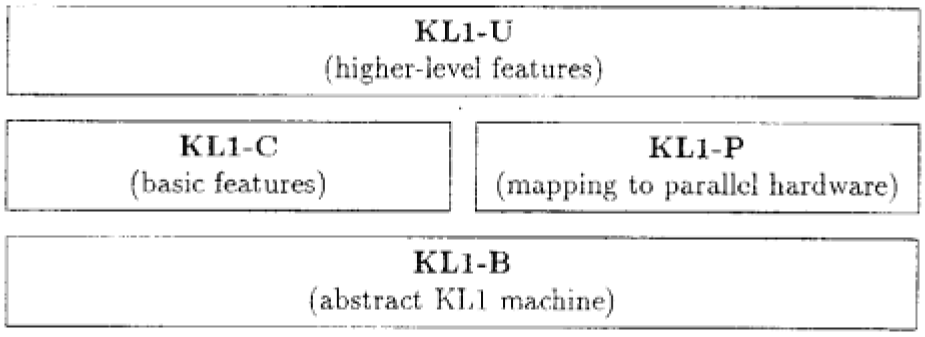

Figure 2: Structure of KL1 (1985)

the coordination of Kazuo Taki, the purpose being to provide an environment for the development of parallel software. Thus, by the end of the initial stage, we could barely establish a starting point of research in the intermediate stage.

5 From GHC to KL1

In June 1985, the intermediate stage of the FGCS project started, and I joined ICOT on loan from NEC.

Shortly before that, the KL1 Design Task Group (the members being Furukawa, Takeuchi, Miyazaki, Ueda and Tanaka at that time) prepared a revised internal report on KL1. The two main aspects of the revision were (i) the adoption of GHC in view of parallel execution and (ii) the reallocation of proposed features to three sublanguages, KL1-C (core), KL1-P (pragma), and KL1-U (user). KL1-C, the core part of KL1, was supposed to be GHC augmented with some metacall construct to support meta-inference and modularization. KL1-P was supposed to deal with the mapping between KL1-C programs and physical resources of the underlying implementation. The proposed component of KL1-P were an abstract machine model, a mechanism for allocating goals to processors, and a mechanism for scheduling goals allocated in the same processor. KL1-U was considered as a set of higher-level language constructs to be compiled into KL1-C and KL1-P, which included the support of pure Prolog (with a set/stream interface) and a module construct.

Another sublanguage, KL1-B, was added to KL1 after a while. Although KL1-C and KL1-P were supposed to be the lowest-level sublanguages for programmers, they were still too high-level to be executed directly by hardware. We decided to have a layer corresponding to the Warren Abstract Machine for Prolog. Initial study of the operating system for PIM, called PIMOS, started as well in June 1985.

We had assumed that KL1-C had all the features of GHC, including nested guards, until Miyazaki and I visited Shapiro's group at the Weizmann Institute for two weeks in July–August 1985. During the visit, we had intensive discussions on the differences between GHC and Concurrent Prolog/FCP. We had discussions also on the subsetting of GHC to Flat GHC, an analogue of FCP obtained from GHC.

People at the Weizmann Institute (Stephen Taylor in particular, who co-designed

Strand and PCN later) were greatly interested in Flat GHC as an alternative to FCP. However, they were concerned that the smaller atomic operations of Flat GHC made the language less robust for describing their Logix operating system. In Concurrent Prolog and FCP, a goal publishes binding information to outside *upon* the reduction of a goal to others, while in GHC, the publication is done *after* reduction using an independent unification goal in a clause body. The separation made implementation much easier, but caused a problem in their meta-interpreter approach to operating systems in which the failure of a unification body goal might lead to the failure of the whole system.

Our visit provoked many discussions in the FCP group, but they finally decided not to move to Flat GHC on the ground that Flat GHC was too fragile for the meta-interpreter approach [17]. On the other hand, we chose the metacall approach because we thought that the meta-interpreter approach would require very careful initial design in order to get everything work well, which could take too much time for us. The metacall approach was less systematic, but this meant that it would be easier to make extensions if they became necessary in the development of the PIMOS operating system.

Back in ICOT, a meeting was held to discuss whether we should move from GHC to Flat GHC. Since Flat GHC was clearly preferable from an implementation point of view, the question was whether the OR-parallel execution of different nested guards was really necessary, or it could be efficiently compiled into the AND-parallel execution of different body goals. We did not have a definite answer, but decided to start with Flat GHC since nobody claimed the necessity of nested guards. A week later, Miyazaki submitted the detailed design of a Flat GHC implementation for Multi-SIM, and Taki submitted the design of interconnection hardware for Multi-SIM. Miyazaki also submitted a draft specification of KL1-C as a starting point for discussions. The points to be discussed included the detailed execution rule of guards, distinction between failure and suspension, the detail of metacall predicates, the treatment of extralogical predicates, requirements for systems programming, and the handling of various abnormal situations (i.e., exception handling).

However, the detail of KL1-C was left unfinalized until summer 1987; we had a number of things to do before that. From an implementation point of view, what we had to do first was to develop basic technologies for the parallel implementation of Flat GHC, such as memory management and distributed unification. From a programming point of view, we had to accumulate experiences with Flat GHC programming. Although the focus of the R&D of parallel implementation was converged on (Flat) GHC by the end of 1985, it was still very important to accumulate evidences, particularly from the programming point of view, that the concurrent logic programming approach was really feasible. One of the tallest hurdles to be cleared in that respect was to establish how to program search problems in Flat GHC.

I started to work on compilation from pure Prolog to Flat GHC in spring 1985. Since Hideki Hirakawa had developed a pure Prolog interpreter in Concurrent Prolog[9], the initial idea was to build its compiler version. However, the interpreter used an extralogical feature—the copying of non-ground terms—which turned out not to make any sense in the semantics of GHC. After some trial and error, in September 1985, I came up with a new method of compiling a subset of pure Prolog into Flat GHC programs that enumerated the solutions of the original programs. The technique was not as general as people wanted to be in that it required the mode analysis of the original programs, but the avoidance of the extralogical feature lead to higher performance as well as clearer semantics.

Although the technique itself was not widely used later, people started to agree that an appropriate compilation technique could generate efficient concurrent logic programs for search problems. An important outcome along this line was MGTP (Model Generation Theorem Prover) for a class of non-Horn clause sets [8].

My main work in 1985 and 1986 was to examine and justify the language design from various respects and thus to make the language more robust. I had a number of opportunities to give presentations and have discussions on GHC, which were very useful to improve the way in which the language was explained. The paper on GHC was first presented at the Japanese Logic Program Conference in June 1985 [20]. A two-day tutorial on GHC programming, with a textbook and programming assignments, was held in May 1986 and 110 people attended. All these activities were quite important, because people had not been well exposed to actual GHC programs and had little ideas about how to program things in GHC.

At first, I was introducing GHC to people by comparing it with the framework of logic programming. However, I started to feel that it was better to introduce GHC as a model of concurrent computation. GHC looked like an concurrent assembly language as well, which featured process spawning, message sending/receiving, and dynamic memory management. The description of the syntax and the semantics of GHC for presentation was finally arranged in one transparency, where I referred to the syntactic constructs of first-order logic or Prolog only for conciseness.

KL1-related R&D activities in the intermediate stage started as collaboration between the first research laboratory for basic research (to which the KL1 Design Task Group belonged) and the fourth research laboratory for implementation. As the Multi-SIM project proceeded, however, the interaction between the two laboratories became smaller. The fourth research laboratory had to design the details of the implementation, while the the first research laboratory was concerned with various topics related to concurrent and parallel programming. In November 1986, all the development efforts related to KL1, including the design of KL1, were gathered in the fourth research laboratory.

The detail of KL1 had to be determined with many practical considerations in implementation. The role of GHC was like pure Prolog in (ordinary) logic programming, and there was still a gap down to the kernel language for real parallel hardware. I was of course interested in the design of KL1, but thought that there would be no other choice than to leave it to the implementation team. Around 1986, I had to spend much of my time in giving tutorials on GHC and writing tutorial articles. I did have another implementation project with Masao Morita, but it was rather a research project with the purpose of studying the relationship between language specification and sophisticated optimization techniques.

It was summer 1987 that Chikayama and his team finally fixed the design of KL1-C and KL1-P. The design of KL1-C reflected many discussions we had since Miyazaki's draft specification, and took Chikayama's MRB scheme (memory management scheme based on 1-bit reference counting) [4] into account. KL1-C turned out not to be a genuine extension of Flat GHC but had several ad hoc restrictions which were mainly for implementation reasons. I did not like the discrepancy between pure and practical versions of a language, but I felt that if some discrepancy was unavoidable, the best way was for separate persons to design them. In our project, both GHC and KL1 were important in their own rights and had different, rather orthogonal design rationales which were not to be confused. Fortunately, the discrepancy was far smaller than the discrepancy between pure Prolog and Prolog, and could be negligible when discussing the fundamental differences between GHC and KL1 (see Section 7.1).

6 Research in the Final Stage

Since 1987, the activities related to the kernel language in the first research laboratory were focused on basic research on Flat GHC and GHC programming. The additional features of KL1 (by KL1 we mean KL1-C and KL1-P henceforth, ignoring the upper and lower layers) were too practical for theoretical study, and Flat GHC itself still had many aspects to be explored, the most important of which were formal semantics and program analysis.

I had long thought that a kernel language must reconcile theory and practice (and people working on theory and those on practice) for its own "healthiness", in addition to reconciling parallel architecture and knowledge information processing. A programming language, particularly a declarative language, can easily split into a version for theoretical study and another version for practice, between which no substantial relationship remains. I wanted to avoid such a situation. Unfortunately, the interests of most ICOT theoreticians were not in concurrent logic programming, with a few exceptions including Masaki Murakami who worked on the semantics of Flat GHC and Kenji Horiuchi who worked on abstract interpretation. Since January 1988, I also started to think how the set of unfold/fold transformation rules for Flat GHC, initially proposed by Furukawa, should be justified. I finally came up with something like an asynchronous version of theoretical CSP, in which each event was a unit transaction between an observee process and its observer, and presented it at the FGCS'88 conference.

In the FGCS'88 conference, I was invited to the final panel discussion on "theory and practice of concurrent systems" chaired by Shapiro, and presented my position on the role and the future direction of kernel languages [21]. The panel was exceptionally well organized and received favorable responses (as was unusual with panel discussions).

I suggested two research directions of KL1 in the panel. The first was the reconstruction of meta-level features in KL1, where by meta-level features I meant the operations that referred to and/or modified the "current" status of computation. Jiro Tanaka was interested in reflection since 1986 and was designing the reflective features for Flat GHC with his colleagues. I liked the approach, but felt that a lot of work was necessary until we could build a full-fledged concurrent system with reflective operations.

The second was the simplification of KL1 and the development of sophisticated optimization techniques, the motivation being to promote KL1 programming with many small concurrent processes. The ultimate goal was to make (a certain class of) processes and streams as efficient as records and pointers in procedural languages. I became interested in optimization techniques for processes that are almost always suspending, and started a study with Masao Morita since September 1988. The work was intended to complement the R&D of Multi-PSI and PIM and to explore the future specification of KL1 beyond the FGCS project.

We soon came up with the basic idea of what we later called the message-oriented implementation technique [23], though it took a long time to generalize it. We found it interesting that Flat GHC programs allowed an implementation technique totally different from the one adopted by all the other implementations.

Sophisticated optimization clearly involved sophisticated compile-time analysis of programs, particularly the analysis of information flow (mode analysis). Concurrent logic languages employed unification as the basic means of communication. Although mathematically elegant, its bi-directionality made the distributed implementation rather complicated. From the language point of view, the bi-directionality might cause unification failure, the failure of unification body goals. Unification failure was considered an exceptional phenomenon analogous to division-by-zero in procedural languages (not just an analogy, as explained in [24]), and hence it was much more desirable to have a syntactic means to avoid it than to have it processed by an exception handler.

On the other hand, people working on development were skeptical about program analysis, suspecting that it was not practical for very large programs. The skepticism, however, lead me to develop an efficient mode analysis technique that was efficient and amenable to separate analysis of (very) large programs [23]. The technique was based on a mode system which turned Flat GHC into a strongly moded subset called *Moded Flat GHC*. I presented the technique in ICOT's 1990 new-year plenary meeting. Very interestingly, two other talks at the meeting argued general unification in KL1 as well. The group implementing distributed unification complained of its complexity. The the group working on natural languages and knowledge representation pointed out that unification in KL1 did not help in implementing unification over richer structures like feature graphs. These arguments made me convinced that general unification was not necessary or useful *at the kernel language level*, though the KL1 implementations on PIM had made too much progress to stop implementing general distributed unification. KL1 implementations on PIM would have been considerably simpler if the mode analysis technique had been proposed earlier.

7 Reflections and Future Prospects

7.1 GHC and KL1

How close is the current status of KL1 to my vision?

In many senses, KL1 was designed from very practical considerations, while the main concern of GHC was the basic framework of concurrent computation. As a positive aspect of KL1's design policy, its performance is no worse than procedural languages in terms of computational complexity, and the absolute performance is also pretty good for a novel symbolic processing language.

On the other hand, the constructs for meta-programming have stayed rather conservative. I expected that practical meta-programming constructs with some theoretical background could be designed finally, but it turned out to be very difficult. Also, the precise semantics of guards seems to have somewhat ad hoc aspects. For instance, the *otherwise* construct for specifying 'default' clauses could have been introduced in a much more controlled way that allowed better formal interpretation.

From a methodological point of view, the separation of the two languages, GHC and KL1, resulted in success [25]. In designing these two languages, it turned out that we were trying to separate two different, though closely related notions: concurrency and parallelism. Concurrency has to do with correctness, while parallelism has to do with efficiency. GHC is a concurrent language, but its semantics is completely independent from the underlying model of implementation. Before GHC was designed, Shunichi Uchida, who lead the implementation team, maintained that the basic computational model of KL1 should not assume any particular granularity of underlying parallel hardware.

To make effective use of parallel computers, we should be able to specify how a program should most desirably be executed on them—at least when we wish. However, the specification tends to be implementation dependent and is best given separately. This is an important role of KL1(-P). The clear separation of concurrency and parallelism made it easier to tune programs without affecting their meaning.

On GHC, the main point of success is that it simplified the semantics of guards by unifying two previously distinct notions: synchronization and the management of binding environments When Gérard Huet visited ICOT in 1988, he wrote a CAML implementation of Flat GHC in a few days. I was impressed with the quick, constructive way of understanding a programming language he took, but this was possible because GHC was so small.

Another point of success is that GHC turned out be very stable—now for eight years. I always emphasized the design principles and basic concepts of GHC whenever I introduced it, and stubbornly kept the language unchanged. This may have caused frustration to GHC/KL1 programmers. Indeed, the design of GHC has not been considered deeply from a software engineering point of view. However, the essence of GHC is in its semantics; the syntax could be re-designed as long as a program in the new syntax can be translated to a program in the current syntax in a well-defined manner. I found the design of user languages much harder to justify, though they should be useful for the development of large software. Many candidates for KL1-U were considered in ICOT, but the current one turned out to be a rather conservative set of additional syntactic conveniences.

Although I have kept GHC unchanged, I have continued research on it. It added much to the stability of the language and improved the way the language was explained. Many ideas which were implicit when GHC was designed were materialized later by the research inside and outside ICOT, and contributed to the justification of the language design. Important theoretical results from outside ICOT include the logical account of the communication mechanism by Michael Maher [12] and Vijay Saraswat's work on concurrent constraint programming [14] that subsumes concurrent logic programming and Flat GHC in particular. On a personal side, I have always been interested in clarifying the relationship between concurrent logic programming and other formalisms of computation, including (ordinary) logic programming and models of concurrency. I have been interested also in subsetting and finally came up with a strongly moded subset called Moded Flat GHC.

A lot of people in the project worked on the implementation of KL1 and KL1 programming, and produced innovative outcomes [1]. They were all important in demonstrating the viability of the concurrent logic programming approach and in getting feedback to implementation and future language design. I believe our R&D of a new paradigm of parallel symbolic programming based on a new programming language went quite well in a promising direction, though of course, many things remain to be done.

Did logic programming have anything to do with the design of KL1? The objective of concurrent logic programming is quite different from the objective of logic programming [22], but still, logic programming played an important role in the design of GHC by giving it strong guidelines. Without such strong guidelines, we may have relied too much on existing concurrency constructs out there and have designed a clumsier language. It is not easy to incorporate many good ideas coherently in a single language.

As a result, GHC programs still allow non-vacuous logical reading. Instead of featuring don't-know nondeterminism, GHC and other concurrent logic languages tried to give better alternatives to operations that had to be done using side effects in Prolog. Logic programming provided a nice framework for reasoning and search and, at the same time, a nice framework for computing with partial information. Concurrent logic programming exploited and extended the latter aspects of logic programming to build a versatile framework of concurrent computation.

Of course, the current status of concurrent logic programming is not without problems. First of all, the term "concurrent logic programming" itself and the fact that it was born from logic programming were—ironically enough—a source of confusion. Many of us considered GHC as an unduly restrictive logic programming language rather than a flexible concurrent language. I had to try to avoid unfruitful controversy on whether concurrent logic programming languages are 'logic' programming languages. Also, largely due to the confusion, the interaction of the concurrent logic programming community with the community of concurrency theory and the community of object-oriented concurrent programming has been surprisingly small. We should have paid more attention to concurrency theory much earlier, and should have talked much more with people working on object-oriented concurrent programming. The only basic difference between object-oriented concurrent programming and concurrent logic programming seems to be whether the notion of a sequence of messages is hidden or exposed as a first-class object.

7.2 ICOT as a Research Environment

ICOT provided an excellent research environment. I could continue to work on language issues for years and could discuss with many people inside and outside Japan, which would have been much more difficult elsewhere. E-mail communication to and from overseas was not available until 1985. Of the three stages of the project, the initial stage (fiscal 1982–1984) was rather different in the sense that it gave us working on KL1 much freedom as well as much responsibility for the R&D of subsequent stages.

I have never felt that ICOT's adherence to logic programming acted as an obstacle to kernel language design; the details were largely up to us researchers, and it was really interesting to try to build a system of new concepts based on logic programming.

The project's commitment to logic programming was liable to be considered extremely political and may have come as an obstacle to some of the researchers who had their own fields of interest outside (concurrent) logic programming. However, in retrospect, ICOT's basic research activities, particularly those not directly related to concurrency and parallelism, could focus even more on logic programming and its connection to other research fields.

Parallelism, too, was not a primary concern for most people working on applications. Parallel programming in KL1 was probably not an easy and pleasant task for them. However, clearly, somebody had to do that pioneering work and contribute to the accumulation of good programming methodologies.

7.3 Position and Beliefs

Fortunately enough, I could keep my position on my research subject very consistent—at least since 1984 when I became acquainted with the project. I was consistently interested in clarifying the relationship and interplay between different concepts rather than amalgamating them. The position, for instance, reflected in the research on search problems in concurrent logic languages. Although the Andorra principle improved much on the previous approaches to amalgamating logic programming and concurrent logic programming, our research on search problems, including the MGTP project, stuck to the compilation approach throughout. A interesting finding obtained independently from my work on exhaustive search and the MGTP work is that a class of logic programs, which the specialists call *range-restricted*, is fundamentally easier to handle than others. Thus our approach lead us to recognize the importance of this concept.

The separation of a concurrent language GHC and a parallel language KL1 is another example. It is often claimed that GHC is a language suitable for systems programming, but the fact is that GHC itself lacks some important features for systems programming, which are included in KL1. The panel discussion of the FGCS'88 conference had a heated debate on whether to expose parallelism to programmers or to hide them. My position was to expose parallelism, but in a tractable form. This was exactly what KL1 tried to address.

In language design, there has been a long controversy within the concurrent logic language community on whether reduction (of a goal) and unification (for the publication of information) should be done atomically or separately. Here again, we stuck to the separation approach.

One reason why I stuck to the separation of concepts is that the gap between parallel hardware and applications software seemed to be widening and was unlikely to be bridged by a universal single paradigm. The initial approximation to the paradigm was logic programming, but it turned out that we had to devise a system of good concepts and notations. These systems and concepts were supposed to form a new 'methodology,' which the FGCS project was going to establish as its principal objective. GHC and KL1 were to form the substratum of the system. (This is why the performance of KL1 implementations is very important, by the way.) As higher-level concepts and notations, languages like GDCC [1] and Quixote came up later. First-order logic itself can be regarded as one of such higher-level constructs, in the sense that MGTP compiles it to KL1 programs. These languages will play the role of Mandala and KL2 we once planned.

I tried to put myself in between theory and practice; I was interested in their interaction. Now I am quite confident that a language designer should try to pay attention to various aspects including its definition, implementation, programming and foundations simultaneously. Language design is the reconciliation of constraints from all these aspects. (In this sense, our approach to the project was basically, but not strictly, middle-out.) Mode analysis and the message-oriented implementation technique were the recent example where the simultaneity worked well. It would have been very difficult to come up with these ideas if we had pursued theory and practice separately. In the combination of high-level languages and recent computer architectures, sophisticated program analysis plays an important role. It is highly desirable that such analysis can be done systematically rather than in an ad-hoc manner, and further that a theory behind the systematic approach is expressed naturally in the form of a language construct. By stipulating the theory as a language construct, it becomes a concept sharable among a wider range of people.

Language designers need feedbacks from specialists in related fields. In semantics research, for instance, one position would be to give precise meanings to given programming languages, but it would be much more productive if the mathematical formulation yields constructive feedback back to language design.

7.4 The Future

What will the future of GHC/KL1 and concurrent logic programming in general be? Let us look back to the past to predict the future.

The history of the kernel language design was the history of simplification. We moved from Concurrent Prolog to GHC, and from GHC to Flat GHC. Most people seemed to believe we should implement distributed unification for Flat GHC at first, but I am now very inclined not to do so. The simplification needed a lot of discussions and experiences, but the performance requirement has always been a strong thrust to this direction. It is not yet clear whether we can completely move to Moded Flat GHC in the next system, but if successful in moving, I expect the performance can be around half of the performance of comparable programs written in procedural languages. The challenge is to achieve the performance not in an ad-hoc manner:

> [20, Section 5.3] *For applications in which efficiency is the primary issue but little flexibility is needed, we could design a restricted version of GHC which allows only a subclass of GHC and/or introduces declarations which*

help optimization. Such a variant should have the properties that additional constructs such as declarations are used only for efficiency purposes and that a program in that variant is readable as a GHC program once the additional constructs are removed from the source text.

[20, Section 9] *We hope the simplicity of GHC will make it suitable for a parallel computation model as well as a programming language. The flexibility of GHC makes its efficient implementation difficult compared with CSP-like languages. However, a flexible language could be appropriately restricted in order to make simple programs run efficiently. On the other hand, it would be very difficult to extend a fast but inflexible language naturally.*

Review of the design of KL1 is now very important. The design of different models of PIMs may not be optimal as KL1 machines, because they had to be designed when we did not have enough knowledge about KL1 implementation and KL1 programming. Also, as experimental machines, they included various ideas we wanted to try. Now the machines were built and almost a million lines of KL1 programs have been written. Based on the experience, we should try to simplify the language and the implementation with minimum loss of compatibility and expressive power.

Another problem KL1 has to face is the huge economical and social 'inertia' on the choice of programming languages. Fortunately, the fact that KL1 and other concurrent logic languages address the field of parallel computing makes things more advantageous. For example, PCN [3], an descendant of concurrent logic languages, addresses an important issue: parallelization of procedural programs. I am glad to see that a new application area of concurrent logic programming is developed this way, but at the same time, I feel that we should study whether parallel applications in concurrent logic languages can be made to run very efficiently without interfacing to procedural codes.

Formal techniques, such as verification, are the area where the progress of our research has been very slow so far. However, we believe that GHC/KL1 is quite amenable to formal techniques compared with other concurrent languages. The accumulation of technologies and experiences should be done steadily, as the history of Petri nets shows.

In his invited lecture of the final day of the FGCS'92 conference, C. A. R. Hoare concluded his talk, titled "Programs Are Predicates" [1] with comments on the similarities between his and our approaches to programming languages and formalisms, listing a number of keywords—simplicity, efficiency, abstraction, predicates, algebra, concurrency, and non-determinism.

Acknowledgments

The author is indebted to Akikazu Takeuchi for his comments on the early design activities of KL1 (Section 3.1).

References

[1] ICOT (ed.), *Proc. Fifth Generation Computer Systems 1992*. Ohm-sha, Tokyo, 1992.

[2] Clark, K. and Tärnlund, S. -Å. (eds.), *Logic Programming*, Academic Press, London, 1982, pp. 153–172.

[3] M. Chandy and S. Taylor, *An Introduction to Parallel Programming*. Jones and Bartlett Pub., Inc., Boston, 1992.

[4] Chikayama, T. and Kimura, Y., Multiple Reference Management in Flat GHC. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 276–293.

[5] Clark, K. L. and Gregory, S., PARLOG: A Parallel Logic Programming Language. Research Report DOC 83/5, Dept. of Computing, Imperial College of Science and Technology, London, 1983.

[6] Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, London, 1984. Also in *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.

[7] Furukawa, K., Kunifuji, S., Takeuchi, A. and Ueda, K., The Conceptual Specification of The Kernel Language Version 1. ICOT Technical Report TR-054, ICOT, Tokyo, 1984.

[8] Fujita, H. and Hasegawa, R., A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm. In *Proc. Eighth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 535–548.

[9] Hirakawa, H., Chikayama, T. and Furukawa, K., Eager and Lazy Enumerations in Concurrent Prolog. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, 1984, pp. 89–100.

[10] Hirata, M., Letter to the editor. *Sigplan Notices*, Vol. 21, No. 5 (1986), pp. 16 17.

[11] Hoare, C. A. R., Communicating Sequential Processes. *Comm. ACM*, Vol. 21, No. 8 (1978), pp. 666–677.

[12] Maher, M. J., Logic Semantics for a Class of Committed-Choice Programs. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, Cambridge, MA, 1987, pp. 858–876.

[13] Nakashima, H., Knowledge Representation in Prolog/KR. In *Proc. 1984 Symp. on Logic Programming*, IEEE Computer Society, 1984, pp. 126–130.

[14] V. A. Saraswat and M. Rinard, Concurrent Constraint Programming (Extended Abstract). In *Conf. Record of the Seventeenth Annual ACM Symp. on Principles of Programming Languages*, ACM, 1990, pp. 232–245.

[15] Sato, M. and Sakurai, T., Qute: A Functional Language Based on Unification. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, Tokyo, 1984, pp. 157–165.

[16] Shapiro, E. and Takeuchi, A., Object Oriented Programming in Concurrent Prolog. *New Generation Computing*, Vol. 1, No. 1 (1983), pp. 25–48.

[17] Shapiro, E. Y., Concurrent Prolog: A Progress Report. *Computer*, Vol. 19, No. 8 (1986), pp. 44–58.

[18] Turner, D. A., The Semantic Elegance of Applicative Languages, In *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture*, ACM, 1981, pp. 85–92.

[19] Ueda, K. *Concurrent Prolog Re-Examined.* ICOT Tech. Report TR-102, ICOT, Tokyo, 1985.

[20] Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming '85*, Wada, E. (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, Berlin Heidelberg, 1986, pp. 168-179.

[21] Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, Tokyo, 1988, pp. 582–591.

[22] Ueda, K., Parallelism in Logic Programming. In *Information Processing 89*, G. X. Ritter (ed.), North-Holland, 1989, pp. 957–964.

[23] Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 3–17. A revised, extended version submitted to *New Generation Computing*.

[24] Ueda, K., Designing a Concurrent Programming Language. In *Proc. InfoJapan'90*, Information Processing Society of Japan, Tokyo, 1990, pp. 87–94.

[25] K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (Dec. 1990), pp. 494–500).