

Design of the Kernel Language for the Parallel Inference Machine

KAZUNORI UEDA AND TAKASHI CHIKAYAMA

Institute for New Generation Computer Technology, Mita Kokusai building 21F, 4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

We review the design of the concurrent logic language GHC, the basis of the kernel language for the Parallel Inference Machine being developed in the Japanese Fifth Generation Computer Systems project, and the design of the parallel language KL1, the actual kernel language being implemented and used. The key idea in the design of these languages is the separation of concurrency and parallelism. Clarification of concepts of this kind seems to play an important role in bridging the gap between parallel inference systems and knowledge information processing in a coherent manner. In particular, design of a new kernel language has always encouraged us to re-examine and reorganise various existing notions related to programming and to invent new ones.

Received August 1990

1. INTRODUCTION

The Japanese FGCS (Fifth Generation Computer Systems) Project,¹⁴ which is a ten-year project started in 1982, aims at developing methodologies and technologies for supporting knowledge information processing with highly parallel inference machines.

The outstanding feature of the FGCS project is that it takes the middle-out approach, which means designing a novel kernel language that bridges the enormous semantic gap between parallel hardware and application software. Years ago, many people hoped that this gap would be narrowed gradually, but instead it seems to be widening, because applications are becoming more and more sophisticated, while recent development of computer architecture (such as RISC and parallel architectures) requires us to exploit the physical characteristics of programs such as communication locality better. This is not necessarily an undesirable phenomenon for very efficient processing and the understanding of computation, though it incurs more difficulty in bridging the gap.

The approach adopted by the FGCS project is to use the logic programming paradigm as the bridge. Of course, merely proposing a single paradigm does not suffice; that was simply a starting point and we had to materialise the paradigm in the form of a kernel language. One reason why logic programming seemed appropriate as a starting point is that it has no sequentiality concept. Prolog, the most successful outcome from the logic programming paradigm so far, relies more or less on sequentiality, but the basic framework of logic programming seemed to provide us with a good basis for research on parallelism.

Indeed, much work has been done on parallelism in logic programming. This can be classified into two major directions: parallel execution of logic programs without explicit specification of concurrency (which we call ordinary logic programs henceforth), and the design of concurrent logic programming languages. The former direction is concerned with exploiting the power of parallel computers in a way transparent to programmers (the most notable example of which is the OR-parallel execution of Prolog), while the latter direction is

concerned with parallelism at the logical level, which we usually refer to as concurrency. We will henceforth use the term parallelism only to mean parallelism at the physical level.

The FGCS project decided to follow the latter direction in principle. We felt that parallel execution of ordinary logic programs was not sufficient to cover all the levels of abstraction between the applications layer and hardware layer. A good concurrency formalism is needed to write reactive systems elegantly,¹² and we decided to design a kernel language based on the concurrent logic programming paradigm. We know that not all applications require the concurrency formalism, but we were quite confident that different program paradigms can be implemented on top of the concurrency framework. Also, concurrent languages provide us with a natural construct through which to consider parallel execution and to design parallel algorithms, namely concurrent processes.

One way to design a concurrent language would be to augment a sequential language with the primitives for communication and concurrency. However, we chose to design our kernel language independently of existing sequential languages. We had long felt that concurrent programming could be made much easier and thus be promoted by finding a simple formalism of concurrency and a good concurrent language. Finding a simple formalism seemed to be useful also for clarifying various concepts related to concurrency and parallelism. The concurrent language we designed is called Guarded Horn Clauses (GHC),^{27,29} which is described in Section 2.

In designing GHC, we completely separated the notion of concurrency and the notion of parallelism, and included only the former in the language constructs. This is because the specification of how a concurrent program should run on a parallel computer tends to be implementation-dependent. To make effective use of parallel computers, however, we should be able to specify how a program should most desirably be executed on them at least when we wish. One may claim that automatic parallelisation is clearly more desirable than explicit control of parallelism from programmers' point of view. However, in order to develop good automatic parallelisation schemes, we should carry out our research

using languages with explicit control of parallelism. Our kernel language, called KL1,³ is based on GHC, but takes the above issue of parallel execution into account among other things. So KL1 can be called a parallel language, while GHC is a concurrent language. The design of KL1 will be described in Section 3.

2. CONCURRENT LOGIC LANGUAGE GHC

The design of the kernel language for the Parallel Inference Machine,¹⁰ the machine we are now building, was started in 1982. The concurrent logic languages we considered as the possible basis of the kernel language include Relational Language,⁴ Concurrent Prolog²³ and PARLOG.^{5,6} In particular, we studied Concurrent Prolog in detail, the most expressive of these languages, and implemented it fully on a general-purpose sequential computer.¹⁷ We had known through our experiences that one of the most useful ways to understand and review a programming language is to try to implement it. Implementation compels us to clarify the details of a language, and if successful, gives us a constructive evidence (though not a complete proof) that the language constructs are well and reasonably defined.

Guarded Horn Clauses (GHC) was born at the very end of 1984 through these studies, the most direct clue being the attempt to clarify the atomic operations of Concurrent Prolog.

GHC shares its basic framework with other concurrent logic languages. First, a GHC program is a set of guarded clauses. Secondly, GHC features no don't-know nondeterminism (built-in search capability) but features don't-care nondeterminism, which allows us to program reactive systems that interact with the outside world. Reactive systems in concurrent logic languages are based on the process interpretation of logic,⁸ in which a goal (or a multiset of subgoals derived from it) is regarded as a process and processes communicate by generating and observing bindings (between shared logical variables and their values). Like most concurrent logic languages, all bindings communicated between GHC processes are determinate, that is, they are never revoked once published to other processes. The determinacy of bindings is essential in reactive systems, because the bindings may be used for interacting with the real outside world.

2.1 The key idea of GHC

What then is the key idea of GHC? As explained above, one important aspect of concurrent logic languages is the determinacy of bindings. In general, the execution of a concurrent logic program proceeds using parallel input resolution that allows parallel execution of different goals, but under the following rules to guarantee the determinacy of bindings.²⁹

(1) The guards (including the heads) of different (guarded) clauses called by a goal g can be executed concurrently, but they cannot instantiate g .

(2) The goal g commits to one of the clauses whose guards have succeeded (see (4) below).

(3) The body of a clause to which g has committed can instantiate g . The bodies of clauses to which g has not committed cannot instantiate g or the guards of the

clauses (this can be achieved simply by not executing them at all).

(4) A goal is said to succeed if it commits to some clause and all its body goals succeed. (Note that the latter half vacuously holds if the body is empty.)

That is, before commitment, a goal can pursue two or more clauses but without generating bindings. After commitment, it can generate bindings but only one clause is left.

Another important aspect of concurrent logic languages is how synchronization is achieved. In general, synchronisation is achieved by restricting information flow caused by unification. Concurrent Prolog uses read-only annotations, and PARLOG uses mode declarations which are used for compiling the unification of input arguments into a sequence of one-way unification and test unification primitives. However, in these languages additional mechanisms are necessary to guarantee restriction (1) above. In Concurrent Prolog, bindings which are generated during the execution of a guard and which would instantiate the caller side are recorded locally, and are published upon commitment. In PARLOG, the guard of a clause C containing a (guard) goal that can instantiate the caller of C is called unsafe, and an additional restriction is imposed that every guard must be safe.¹¹

The key idea of GHC is quite simple. It uses restriction (1) itself as a synchronisation construct. That is, any piece of unification which is invoked directly or indirectly from the guard of a clause C and which would instantiate the caller of C is suspended until it can be executed without instantiating the caller. Thus the safety condition in the sense of PARLOG is automatically satisfied. Moreover, unlike Concurrent Prolog, no bindings need be recorded for later publication. In other words, GHC has integrated two notions: the determinacy of bindings and synchronisation. This conceptual simplification led to GHC being adopted finally as the basis of our kernel language.

Interestingly, the same synchronisation mechanism had been invented independently in the functional language Qute for different purposes.²²

2.2 From GHC to Flat GHC

A kernel language must provide a common framework for people working on various aspects of the project including applications, implementation and theory. Before accepting GHC as the basis of our kernel language, we had to convince ourselves that it satisfies the following conditions.

(1) It is expressive enough.

(2) It can eventually be implemented efficiently, possibly by appropriate subsetting.

(3) It is simple enough to be understood and used by programmers including novices. Also, it is simple enough for theoretical treatment.

It is a social process that a programming language is accepted by a community. It took considerable time and effort until GHC was accepted even within ICOT. The primary reason is that many of us considered GHC as an unduly restrictive logic language rather than a flexible concurrent language. We soon made sure that GHC was expressive enough to write most concurrent algorithms that had been written in other concurrent logic languages,

but that was not enough. How to program search problems was also important, because search problems are a specialty of ordinary logic languages with which our project was started. So we have developed a couple of methods for programming search problems.^{19, 26, 28}

For implementability, we quickly ascertained by rapid prototyping that GHC can be implemented fairly efficiently at least on sequential computers.³²

For simplicity, we continued to study the properties of GHC and looked for a simpler explanation of the language better suited to process interpretation. Now, our interpretation is that a GHC process is an abstract entity which observes and generates information (represented in the form of bindings) and which is implemented by a multiset of body goals. The behaviour of each body goal is defined by guarded clauses that can be regarded as rewrite rules.

A problem with the original definition of GHC is that guard goals do not fit well into the process interpretation. They are most naturally regarded as auxiliary conditions to be satisfied for the rewrite rule containing them to be applied. From a practical point of view, we felt that the expressive power of guard goals did not justify the implementation effort even if it could be implemented efficiently. In short, the generality of guard goals seemed unnecessary.

These considerations led us to reduce GHC to a subset, Flat GHC, a movement inspired by the reduction of Concurrent Prolog to Flat Concurrent Prolog.²⁴ Since Flat GHC arose from rather practical requirements, it did not have a rigorous definition for a long time. The vague idea was that only certain predefined predicates could be called from clause guards, but it was not defined what properties should be satisfied by those predefined predicates. Later on, we became convinced that the sufficient conditions to be satisfied by a guard goal as an auxiliary condition are that it is deterministic (that is, whether it succeeds or not depends only on its arguments) and that it does not produce any bindings. These conditions can be obeyed by restricting predicates called directly or indirectly from a guard to those defined by unit clauses (possibly virtually in the case of predefined predicates), namely clauses with empty bodies. This restriction simplified the theoretical treatment of GHC such as the operational semantics³¹ and program transformation rules.³³

To summarise, the basic idea of Flat GHC is as follows. A program is a set of guarded clauses that can be regarded as rewrite rules of goals. The guard of a clause specifies what information should be observed before applying the rewrite rule, and the body specifies the multiset of goals replacing the original one. A body goal is either a unification goal of the form $t_1 = t_2$, whose behaviour is language-defined, or a non-unification goal, whose behaviour is user-defined. A unification body goal generates information by unifying t_1 and t_2 , and a non-unification body goal represents the rest of the work and will be reduced further.

2.3 Understanding GHC better

When GHC was first proposed, we were not fully aware of many good properties of the language; they were clarified by later work inside and outside ICOT. One example is the process interpretation of Flat GHC

programs. Another example is a logical characterisation of communication and synchronisation due to Maher.¹⁵ He showed that information communicated by processes can be viewed as equality constraints over terms; that the generation of information can be viewed as the publication of a constraint; and that the observation of information can be modelled as the implication of a constraint by the set of constraints published so far.

Thus we have acquired both algebraic and logical characterisations of the communication mechanism used in GHC, which indicates the robustness of the language construct.

Also, we tried to characterise the atomic operations of GHC. Unlike concurrent Prolog but like PARLOG, the publication of bindings is not done atomically upon commitment of a non-unification goal but eventually after commitment, using a unification body goal that can run in parallel with other goals. This means that commitment in GHC is a smaller and simpler operation than commitment in Concurrent Prolog. Moreover, in GHC the information generated by a unification body goal is not an atomic entity in general. It can be transmitted in smaller pieces, possibly with communication delay.

We have found that this liberal computational model of (Flat) GHC is expressive enough to program co-operating concurrent processes and leaves more freedom to implementation. (Flat) GHC is unfortunately not expressive enough to program processes that may not be cooperative. However, the *shoen* construct of KLI (Section 3.1) takes care of such processes.

Another point to note is that GHC has included control for the correct behaviour of processes but excluded any control for efficient execution. GHC has left the latter to KLI in order to clearly distinguish between the two notions. This contrasts with PARLOG, which features sequential AND that can be used for suppressing parallel execution of body goals. We believe that it is important to learn that synchronisation based on information flow is sufficient for writing correct concurrent programs.

Important topics on theoretical aspects of Flat GHC include the relationship with other theoretical models of concurrency such as CCS¹⁶ and theoretical CSP.¹³ Although concurrent logic languages differ from CCS and CSP in that they are based on asynchronous communication and can be used to program dynamically reconfigurable processes, similar mathematical techniques can be used to formalise them.^{9, 21} In Flat GHC, the notion of a transaction³³ captures the externally meaningful unit of communication that corresponds well to an event in synchronous communication. We have not yet obtained a completely satisfactory formal semantics, but we are fairly confident that Flat GHC is theoretically simple enough, while it can be used for practical programming without any modification.

An issue that has always been of great interest since various concurrent logic languages were proposed is how to relate them to ordinary logic languages with don't-know nondeterminism. Our consistent position has been to clarify the difference of these two families of languages and to integrate them with a carefully designed interface.³⁰ In developing a compilation method from ordinary to concurrent logic languages,²⁸ we tried to clarify what it means to 'collect' all solutions of a search

program, which is related to the semantics of all-solutions predicates in Prolog such as `bagof`.

3. PARALLEL LANGUAGE KL1

As described above, we have designed a concurrent logic language Flat GHC as the basis of the kernel language for parallel inference systems. The descriptive power of the language, however, is not sufficient when efficient program execution is our concern. As Flat GHC programs do not say anything about where (i.e. on which processor) the atomic operations making up a computation should be performed, there are many ways to distribute the operations over available processors. As Flat GHC programs only specify the partial ordering of atomic operations, there are many possible total orderings conforming to it. Some distribution and ordering may be more efficient than others. To make sure in all cases that the distribution and the ordering employed are not far from optimal, we must be able to specify physical details of execution to some extent.

We thus designed a parallel programming language based on the concurrent programming language Flat GHC, in which we can specify in certain detail how a program should be executed. This section describes the outline of this language, named KL1.

3.1 Mapping of computation

Flat GHC programs implicitly express any potential parallelism in the sense that no ordering between atomic operations exists except for the ordering essential for correctness. To faithfully exploit this parallelism might be meaningful on an ideal parallel computer which has an unlimited number of processors and in which inter-processor communication has unlimited throughput and no latency. However, any real hardware has a limited number of processors and the cost of interprocessor communication cannot be neglected. To achieve efficiency, control is required on when and where each atomic operation should be performed. We call this control *mapping* in what follows.

One way to solve the problem is to make a language implementation fully responsible for mapping. The current technology of parallel software, however, does not provide an efficient mapping strategy applicable to all application areas; establishing such technology through experiences with diverse applications is one of the principal goals of the research on parallel inference systems in the FGCS project.

Mapping is often implicit in sequential systems. Suppose there are two methods to solve a problem: method *A* may fail to find a solution in rare cases, but always terminates in a short period of time either with a solution or with a failure signal; method *B* is less efficient but always finds a solution. In such a case, the most efficient sequential strategy is to try *A* first and to try *B* only when *A* is unsuccessful. In sequential systems, such strategic decisions for efficiency are usually not clearly separated from the mandatory ordering for the correctness of programs.

Trying *B* only after *A* may not be the best strategy, however, for parallel systems. Method *A* may not require all the computational resource (such as processors) for its execution. In such a case, method *B* should be tried in

parallel with *A*, as long as it does not interfere with the execution of method *A*. This can be realised by providing an elastic guideline of mapping, namely giving *A* a higher priority than *B*.

Sometimes more sophisticated mapping is desirable. Suppose that there are two methods to solve a problem and that, although at least one is known to find a solution efficiently, we cannot tell which beforehand. In such a case, the best scheduling strategy may be to give both methods approximately the same amount of computational resource. Resource management is thus an important part of an algorithm in parallel computation.

In sequential computer systems and in parallel computer systems as extensions of conventional sequential systems, operating systems are primarily responsible for mapping. This is acceptable as far as application programs are mostly sequential and the mapping strategy is implicitly specified and executed using sequencing. In parallel systems where explicit mapping operations are much more frequently required, invoking the operating system for each mapping operation will incur intolerable overheads.

To solve this problem, we have introduced into KL1 the following features, which are intended to be efficiently implemented.

Shoen. *Shoen** represents a group of goals. This group is used as the unit of execution control, namely the initiation, the interruption, the resumption and the abortion of execution. Exception handling and resource consumption control mechanisms are also provided through this *shoen* construct. The *shoen* construct is an extension of the metacall construct proposed by Clark and Gregory.⁶

Priority. A (body) goal of a KL1 program is the unit of priority control. Each goal has an integer priority associated with it. Each *shoen* keeps the maximum and the minimum priorities allowed for goals belonging to it, and the priority of each goal is specified relative to these. The language provides a large number of logical priority levels, which are translated to physically available priority levels provided by each implementation. If no priority is specified, the priority of the parent goal is inherited.

The priority mechanism can be used for programming speculative computation.^{1,20}

Processor specification. Each (body) goal may have a processor specification, which designates the number of the processor on which to execute the goal. Without this, the goal is executed on the same processor as its parent goal.

This straightforward mechanism provides the basis of research in more sophisticated load-distribution strategies. Actually, several automatic load-distribution strategies have been developed for diverse problems. As the optimal load distribution depends heavily on each problem, no single scheme works universally. Instead, typical schemes are planned to be provided as libraries, from which users can select the most appropriate ones for their problems.

One of the most notable characteristics of the KL1 language is that these priority and processor specifications are separated from concurrency control. We call these specifications *pragmas*. Pragmas are merely guide-

* *Shoen* is a Japanese word corresponding to 'manor' in English.

lines for language implementations and may not be precisely obeyed. The same is true of the controlling mechanism of *shoen*; abortion of computation, for example, may not happen immediately. This relaxation makes distributed implementation much easier.

Pragmas are specified within the program but are clearly distinguished syntactically from other language constructs. Pragmas will never change the correctness of the programs,* though the performance may change drastically. As it is not uncommon that more than half of the effort to develop a program is devoted to the design of appropriate mapping, it is most advantageous that the specification of mapping is syntactically isolated from the rest of the program. In many parallel programming languages, the specification of parallel execution is often mixed up with other language constructs, especially with constructs for concurrency control. A major revision is often required for improving efficiency or for running the program on a different implementation, which is liable to introduce new bugs.

3.2 Keeping up with sequential languages

What criterion is appropriate for comparing parallel algorithms? Assume that a parallel algorithm has sequential execution time $c(n)$ (n being the size of the problem) and average potential parallelism $p(n)$. Then the total execution time by this algorithm on an ideal parallel computer is given by $c(n)/p(n)$. This means that an algorithm with more sequential execution time but with still more parallelism is considered to be a better algorithm on an ideal parallel computer.

This, however, does not hold when the potential parallelism, which may vary over time, can exceed the physically available parallelism. With limited physical parallelism, which is always the case in the real world, a parallel algorithm whose sequential time complexity is worse than that of a known sequential algorithm will be beaten by that sequential algorithm running on a sequential computer for sufficiently large n , no matter what $p(n)$ is.

Thus, when designing a parallel algorithm, we must often consider a hybrid strategy that the algorithm switches to a sequential algorithm when the physically available parallelism is used up.

Pure languages such as pure Lisp and pure Prolog cannot straightforwardly express certain kinds of efficient algorithm due to the lack of the notion of destructive assignment. To overcome this requires optimisation techniques that enable an implementation to make use of the destructive assignment of hardware memory. GHC also is a pure language with the same inherent problem. To write efficient algorithms in these pure languages, we must be able to somehow mimic the efficiency of array operations in conventional languages.

For this reason, KL1 introduced a primitive for updating an array element in constant time without disturbing the single-assignment property of logical variables. The primitive can be used as follows:

```
set_vector_element(Vect, Index, Elem,
                  NewElem, NewVect)
```

* To be precise, the priority specification may be used for guaranteeing certain properties of diverging (i.e. autonomously non-terminating) programs.

When an array *Vect*, an index value *Index* and a new element value *NewElem* are given, the predicate binds *Elem* to the value of the *Index*'th element of *Vect*, and *NewVect* to a new array which is the same as *Vect* except that the *Index*'th element is replaced by *NewElem*.

Because some other goals may still have references to the old array *Vect*, a naive implementation might allocate a completely new array for *NewVect* and copy all but one elements. However, when it is known that no goals other than the above *set_vector_element* goal have references to *Vect*, there will be no problem in destructively updating it. In the actual implementation of KL1, a simplified, efficient version of the reference counting scheme² detects such a situation, in which event the new array *NewVect* is obtained in constant time.

This means that any imperative algorithm can be rewritten in KL1 retaining the same computational complexity, as random access memory can always be emulated using a single-reference array. Of course, allowing only one reference to a data structure can decrease the possibility of parallel execution considerably. However, as stated above, the requirement of the computational complexity must be considered only for the sequential parts of parallel algorithms which are invoked after physically available parallelism is used up.

3.3 Implementation

The most advanced implementation of KL1 currently in use is the one on the Multi-PSI system.²⁵ This experimental parallel inference machine has up to 64 processors of PSI-II¹⁸ connected in grid, attaining the peak performance of around 10 MRPS for list concatenation.† Several Multi-PSIs and the KL1 implementation on them are used in the research and development of parallel application software.

A new implementation under development is for a higher-performance inference machine PIM,¹⁰ which is expected to have up to 512 processing elements and attain more than 100 MRPS of peak performance.

4. CONCLUSIONS

We have reviewed the design of the concurrent language GHC, the basis of the kernel language for the FGCS project, and the design of the parallel language KL1, the actual kernel language we are implementing and using. We have explained why we expose both concurrency and parallelism. Both need to be accessible for some programmers, though they may not have to be exposed to all programmers. When a good amount of parallel application software in KL1 has been accumulated, we should try to find appropriate higher-level language constructs for supporting application programmers, together with their implementation techniques on KL1.

We have been careful in separating concurrency and parallelism because they are separate, though closely related, concepts. Concurrency has to do with correctness, while parallelism has to do with efficiency. This means that the semantics of GHC is independent of the underlying model of implementation, while the semantics

† MRPS is for mega reductions per second. This roughly corresponds to MLIPS (mega logical inferences per second) of Prolog.

of KL1 assumes a particular model of implementation. The formal semantics of KL1 is therefore difficult to describe, but this separation has made GHC simpler from a theoretical point of view.

To mention this separation from the programming point of view, our experience shows that writing correct concurrent programs is not difficult. What is still difficult is to write efficient parallel programs. The operating system PIMOS³ for Multi-PSI and PIM was first developed using a KL1 implementation on a general-purpose sequential machine, but almost no synchronisation bugs bothered us when it was installed on Multi-PSI.

The purpose of our research on the kernel language is not only to design a usable programming language, but also to better understand various concepts related to concurrent, parallel and logic programming. We started our project with the logic programming paradigm and then introduced concurrent logic programming, but we must continue to find many good concepts that systematically bridge the semantic gap between parallel computers and knowledge information processing.

Both GHC and KL1 have room for refinement. For instance, recently we found that a simple mode system based on the notion of constraints can be used for simplifying Flat GHC further³⁴ both in terms of programming and of implementation. As this example indicates, implementation, applications and theory interact with one another in designing a programming language. It is very important for the healthy development of the kernel language that the language is used and reviewed by people working on these diverse areas.

Acknowledgements

We are indebted to all our colleagues, too many to be listed here, who have worked and/or are working with us in designing, implementing, and using the kernel language. Special thanks are due to Koichi Furukawa and Akikazu Takeuchi for initiating the research on concurrent logic programming in ICOT.

REFERENCES

1. F. W. Burton, Speculative computation, parallelism and functional programming. *IEEE Trans. Computers* **C-34** (12), 1190–1193 (1985).
2. T. Chikayama and Y. Kimura, Multiple reference management in Flat GHC. In *Proc. 4th Int. Conf. on Logic Programming*, pp. 276–293. MA, MIT Press, Cambridge (1987).
3. T. Chikayama, H. Sato and T. Miyazaki, Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, pp. 230–251. ICOT, Tokyo (1988).
4. K. L. Clark and S. Gregory, A relational language for parallel programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pp. 171–178. ACM, New York (1981).
5. K. L. Clark and S. Gregory, *PARLOG: a Parallel Logic Programming Language*. Research Report DOC 83/5, Department of Computing, Imperial College of Science and Technology, London (1983).
6. K. L. Clark and S. Gregory, Notes on systems programming in PARLOG. In *Proc. Int. Conf. On Fifth Generation Computer Systems 1984*, pp. 299–306. ICOT, Tokyo (1984).
7. K. L. Clark and S. Gregory, PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems* **8** (1), 1–49 (1986).
8. M. H. van Emden and G. J. de Lucena Filho, Predicate logic as a language for parallel programming. In *Logic Programming*, edited K. L. Clark and S.-Å. Tärnlund, pp. 189–198. Academic Press, London (1982).
9. R. Gerth, M. Codish, Y. Lichtenstein and E. Shapiro, Fully abstract denotational semantics for Flat Concurrent Prolog. In *Proc. Third Annual Conf. on Logic in Computer Science*, pp. 320–335. IEEE (1988).
10. A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto, Overview of the Parallel Inference Machine architecture (PIM). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, pp. 208–229. ICOT, Tokyo (1988).
11. S. Gregory, *Parallel Logic Programming in PARLOG: the Language and its Implementation*. Addison-Wesley, Wokingham, England (1987).
12. D. Harel and A. Pnueli, On the development of reactive systems. In *Logics and Models of Concurrent Systems*, edited K. R. Apt, pp. 477–498. Springer, Heidelberg (1985).
13. C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J. (1985).
14. T. Kurozumi, Present status and plans for research and development. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, pp. 3–15. ICOT, Tokyo (1988).
15. M. J. Maher, Logic semantics for a class of committed-choice programs. In *Proc. Fourth Int. Conf. on Logic Programming*, pp. 858–876. MIT Press, Cambridge, MA (1987).
16. R. Milner, *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, N.J. (1989).
17. T. Miyazaki, A. Takeuchi and T. Chikayama, A sequential implementation of Concurrent Prolog based on the shallow binding scheme. In *Proc. 1985 Symp. on Logic Programming*, pp. 110–118. IEEE (1985).
18. H. Nakashima and K. Nakajima, Hardware architecture of the sequential inference machine PSI-II. In *Proc. 1987 Symp. on Logic Programming*, pp. 104–113. IEEE (1987).
19. A. Okumura and Y. Matsumoto, Parallel programming with layered streams. In *Proc. 1987 Symp. on Logic Programming*, pp. 224–231. IEEE (1987).
20. R. Osborne, Speculative computation in Multilisp. In *Parallel Lisp: Languages and Systems*, edited T. Ito and R. Halstead, pp. 103–137. Lectures Notes in Computer Science 441. Springer, Heidelberg (1990).
21. 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*, pp. 232–245. ACM, New York (1990).
22. M. Sato and T. Sakurai, Qute: a functional language based on unification. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, pp. 157–165. ICOT, Tokyo (1984).
23. E. Y. Shapiro, *A Subset of Concurrent Prolog and Its Interpreter*. Tech. Report TR-003, ICOT, Tokyo (1983).
24. E. Y. Shapiro, Concurrent Prolog: a progress report. *Computer* **19** (8), 44–58 (1986).
25. Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki, A load balancing mechanism for large scale multiprocessor systems and its implementation. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, pp. 978–986. ICOT, Tokyo (1988).
26. H. Tamaki, Stream-based compilation of ground I/O Prolog into committed-choice languages. In *Proc. 4th Int. Conf. on Logic Programming*, pp. 376–393. MIT Press, Cambridge, MA (1987).
27. K. Ueda, Guarded Horn Clauses. In *Logic Programming '85*, edited E. Wada, Lecture Notes in Computer Science 221, pp. 168–179. Springer-Verlag, Heidelberg (1986).
28. K. Ueda, Making exhaustive search programs deterministic. *New Generation Computing* **5** (1), 29–44 (1987).

29. K. Ueda, Guarded Horn Clauses: a parallel logic programming language with the concept of a guard. In *Programming of Future Generation Computers*, edited M. Nivat and K. Fuchi, pp. 441–456. North-Holland, Amsterdam (1988).
30. K. Ueda, Parallelism in logic programming. In *Information Process 89*, edited G. X. Ritter, pp. 957–964. North-Holland, Amsterdam (1989).
31. K. Ueda, Designing a concurrent language. To be presented at InfoJapan '90, Information Processing Society of Japan, Tokyo (1990).
32. K. Ueda and T. Chikayama, Concurrent Prolog compiler on top of Prolog. In *Proc. 1985 Symp. on Logic Programming*, pp. 119–126. IEEE (1985).
33. K. Ueda and K. Furukawa, Transformation rules for GHC programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, pp. 582–591. ICOT, Tokyo (1988).
34. K. Ueda and M. Morita, A new implementation technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, pp. 3–17. MIT Press, Cambridge, MA (1990).

Announcements

3–6 SEPTEMBER 1991

VLDB '91: 17th International Conference on Very Large Databases

Barcelona, Catalonia, Spain

Call for Papers

The Conference

VLDB conferences are a truly international forum for identifying, encouraging and exchanging ideas and experiences on research, development and novel applications of database management systems and techniques. Like its predecessors, the seventeenth VLDB Conference aims at bringing together researchers, developers and users of database management systems from academia and industry to share this information and explore new and challenging issues.

Topics

- Logic, deductive and knowledge-based systems
- Extensible, temporal and multi-media databases
- Active and real-time database systems
- Advanced applications and requirements
- Object-oriented and semantic databases
- Data models, languages and user interfaces
- Data structure, access methods and complex objects
- Database theory and algorithms
- Query optimisation, database design and performance
- Storage management, database machines and parallelism
- Database integrity and security
- Concurrency control, transaction processing and recovery
- Distributed and heterogeneous databases

Papers

Original papers of up to 5,000 words in length are invited on topics including, but not limited to, those listed above. All submitted papers will be read and carefully evaluated by the Programme Committee. To submit papers, send five (5) copies of the double-spaced manuscript, including keywords, in English to one of the Programme Co-Chairs by 15 February 1991:

Professor Dr Amílcar Sernadas, INESC, Rua Alves Redol, 9, 70, Apartado 10105, P-1017 Lisboa Codex (Portugal). Internet (US): inesc!acs%solo@relay.EU.net; (Europe): asc%solo@inesc.uucp or acs@inesc.ctt.pt.

Dr Guy M. Lohman, IBM Almaden Research Center, Department K55, Building 801, 650 Harry Road, San Jose, CA 95120–6099, U.S.A. Internet: lohman@ibm.com; BITNet: lohman@almaden.

Panels

Proposals for panel sessions should be sent by 30 March 1991 to:

Professor Alfonso F. Cardenas, University of California at Los Angeles, Department of Computer Science, 3731 Boelter Hall, Los Angeles, CA 90024, U.S.A. Internet: cardenas@cs.ucla.edu.

Tutorials

Proposals for tutorials should be sent by 6 January 1991 to:

Professor Antoni Olive, Facultat d'Informàtica, Universitat Politècnica de Catalunya, Pau Gargallo, 5, E-08028 Barcelona, Catalonia, Spain. Ean: olive@lsi.upc.es. From BITNet: olive%fib.upc.es@cernvax.

Important dates

6 January 1991	Tutorial deadline
15 February 1991	Paper submission deadline
30 March 1991	Panel session deadline
29 April 1991	Notification of acceptance
31 May 1991	Camera-ready copies deadline
3–6 September 1991	Conference in Barcelona

The 17th International Conference on Very Large Data Bases is organised by the Asociación de Técnicos de Informática (ATI, member of FESI) and the Universitat Politècnica de Catalunya on behalf of the VLDB Endowment, in cooperation with the IEEE

Computer Society. ACM cooperation pending.

General Conference Chair

Professor Felix Saltor, Universitat Politècnica de Catalunya, Barcelona.

European Conference Chair

Dr Georges Gardarin, INRIA and Université Paris VII, Le Chesnay and Paris.

U.S. Conference Chair

Dr David K. Hsiao, Naval Postgraduate School, Monterey, California.

VLDB Endowment Representative

Dr Peter Lockemann, Universität Karlsruhe, Karlsruhe.

Local Organising Committee Chair

Dr Jordi Guinovart, IBM Spain, Barcelona.

The 17th VLDB Conference will be held in Barcelona, Catalonia, Spain. Barcelona, settled since the sixth century B.C., is the capital city of Catalonia, a country established in the tenth century. It has been selected to host the next Olympic Games for good reasons: its situation by the Mediterranean Sea, its climate, communications, facilities and cultural events. The Gothic quarter of Barcelona and the Romanesque paintings at the Museum of Catalan Art are world-famous, but visitors to Catalonia can find works of art of different cultures, from Iberian and Roman, through Visigoth and Jewish, to Modern Style – such as Gaudí's Sagrada Família church featured in the Conference logo – and contemporary art: Picasso, Miró, Dali, Tàpies. Barcelona is also known for its fine restaurants and a wide variety of sports activities.

The social programme is planned to include a banquet in a medieval building, and optional trips to the beaches of the Costa Brava and to a winery.

For further information contact:

VLDB '91, DIFINSA, Av. República Argentina, 63, E-08023 Barcelona, Catalonia, Spain.