

# Concurrent Logic Programming Before ICOT: A Personal Perspective

Steve Gregory  
Department of Computer Science, University of Bristol

August 15, 2007

The story of the Japanese Fifth Generation Computer Systems (FGCS) project, and the role of concurrent logic programming in it, has been told many times. In this article, I thought it might be interesting instead to focus on the earliest, most volatile, period in the history of concurrent logic programming: up to about the end of 1984.

## Concurrency, logic programming, and me

My first encounter with concurrency came in about 1977, while developing multi-user accounting software. A client had complained about their system intermittently crashing and corrupting data, and the fault had been traced to a “deletion program” that I had written. I checked my code and soon found the source of the problem, but couldn’t think of a solution, and reported this to my boss. “Nothing is impossible!” was his immediate reaction, but he was also unable to solve the problem. Neither he, an eccentric mathematician, nor I, a naïve junior programmer, had ever heard of mutual exclusion, and we certainly didn’t know how to implement it in Fortran. Apparently, nobody had anticipated this before the (very expensive) system had been implemented and delivered.

I discovered logic programming about two years later, as an undergraduate at Imperial College (IC). By chance, Bob Kowalski had become my personal tutor and I asked his advice about projects. He gave me the main logic programming papers from the 1970s and a draft copy of his book, *Logic for Problem Solving*. The procedural interpretation of logic was completely new to me and looked pretty interesting.

When I started my undergraduate project, it was supervised by Keith Clark. He and Frank McCabe had already created IC-Prolog [4], which was intended to be a more declarative version of Prolog, inspired by Bob’s principle,

Algorithm = Logic + Control

The most interesting feature of IC-Prolog was its *dataflow coroutining*: by annotating a variable with ‘^’ or ‘?’, the containing goal would become a *lazy producer* or *eager consumer* of the variable’s binding. They were also planning to add a *pseudo-parallel* form of conjunction. All of these allowed programs to be defined with simple logic and a control strategy that transferred between goals at run time. My task was to transform coroutining IC-Prolog programs into regular Prolog programs [9].

## Lazy Lisp and Moore's Law

Two trends in the late 1970s influenced the subsequent development of concurrent logic programming. One was the emergence of novel evaluation strategies for functional languages: notably lazy Lisp [6], which had clearly influenced IC-Prolog, and the nondeterministic *frops* function [7] later proposed by the same authors. These demonstrated that declarative languages could be more versatile than they seemed at first glance.

The second factor was a widespread belief that the limit to Moore's Law would soon be reached: the only way for computers to run faster in future would be to build parallel architectures, but how would they be programmed? Declarative languages, with their lack of side-effects, seemed ideal for this. A range of fine-grained parallel architectures sprang up, especially dataflow and functional architectures. There were also a few proposals for exploiting parallelism in logic programs.

The concept of *stream parallelism* — evaluating a consumer and producer in parallel by incrementally producing a data structure — originated with Kahn and MacQueen [11] in the context of functional programs. This was seen as an important source of parallelism in functional programs, but took longer to reach logic programming because of its nondeterminism. At the end of the decade, the idea was investigated by van Emden and de Lucena [5] for deterministic logic programs. At the same, stream (pseudo-)parallelism was added to IC-Prolog, though this was more a form of coroutining than a source of parallelism.

## The first concurrent logic programming language

While working on the transformation of IC-Prolog programs in 1979-80, I was looking for examples of programs using the proposed pseudo-parallel feature. That was when I came across Hoare's CSP paper [10], which proved to be a rich source of ideas. I found that CSP programs could be rewritten elegantly in (a subset of) IC-Prolog. Concurrent programming seemed like a much more exciting application area for logic programming than the usual "*John likes Mary*".

That summer, I took over the implementation of IC-Prolog from Frank. For someone who knew nothing about implementing Prolog, this was a traumatic experience: IC-Prolog was far more complicated than regular Prolog. My main task was to implement pseudo-parallelism. Fortunately, I was already familiar with its operational semantics, but the combination of dataflow coroutining, pseudo-parallelism, and backtracking was a nightmare to implement.

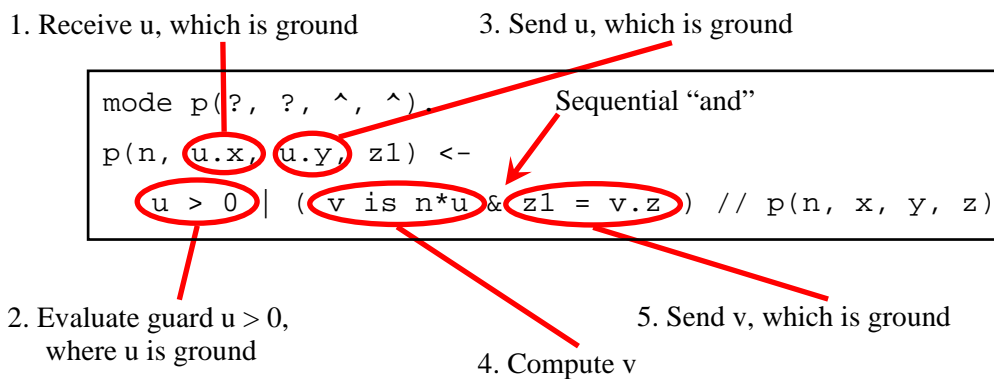
Later in 1980 I followed Keith to Syracuse to act as a teaching assistant for his course (on IC-Prolog) and carry on the research. I wanted to design a logic programming equivalent of CSP, and IC-Prolog seemed just too complex. It seemed reasonable to replace both coroutining and pseudo-parallelism by "real" concurrency, using CSP-like input (*match*) and output (*bind*) operations on channels (shared variables). This was subtly different from what IC-Prolog had, but seemed more practical. Backtracking was more controversial: nondeterminism was considered a key feature of logic programming but was not found in CSP, or any other language that I knew of. Eventually, we decided to get rid of ("don't know") nondeterminism and replace it by CSP-like guards. By March 1981, back in London, we had

completed the definition of what became known as the *Relational Language* [1]. Because of its incompleteness, we didn't dare to call it a logic programming language.

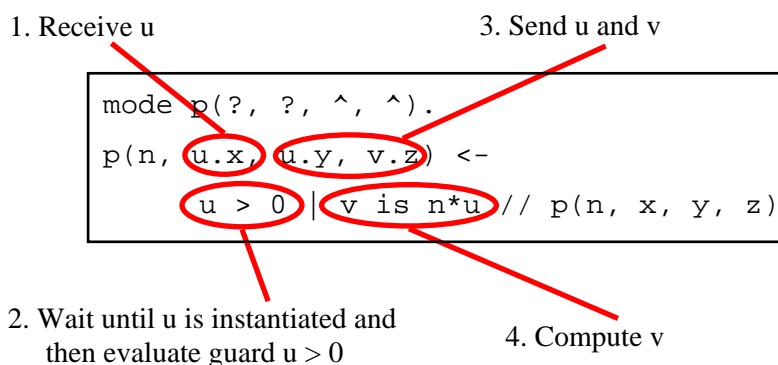
Backtracking was not the only logic programming feature missing from the Relational Language. Variables also had a diminished role, because the modes (used to distinguish input from output arguments) were *strong*: a goal to generate a data structure had to generate the entire structure. Finally, lists had a special significance: a list represented a sequence of messages while any other data structure represented a single message. Message passing was achieved by binding a shared variable to a ground term or a list whose head was ground. These restrictions were to allow the language to be implemented on a distributed architecture.

## The lull before the storm

The next two years were relatively uneventful. My job was to implement and apply the Relational Language. I filled some holes in the language definition, worked out the details of the mode analysis, and generalized the language to allow incremental communication of any term. Originally, a shared variable had to be a list of ground terms, which had the nice property that we could rely on guards being ground. But it also meant that messages had to be completely constructed before sending them; this relied on sequencing, as illustrated here:



It turned out to be more natural to allow a message to be sent before being constructed, like this:



We could no longer rely on ground guards, but it was not a problem because guards could suspend if necessary, just like input matching. Because of the strong modes, guards could be prevented from binding variables in the calling goal, and so the guards were still mutually independent.

I wrote a compiler for the Alice machine, a parallel functional architecture being designed by Mike Reeve and John Darlington at IC. Set constructors, to interface between the language and a Prolog-like language, were added to the language: a feature taken from IC-Prolog. Functions (eager and lazy) were added, mainly because they were easy to implement on Alice, but also because they could be used for set constructors.

The week of October 18-22, 1981 turned out to be significant. By coincidence, this was the week of the first FPLCA conference, where we presented the Relational Language, *and* the first FGCS conference, which discussed the project that was launched in April 1982 with the opening of ICOT. In 1981, I had no idea what the Fifth Generation was, but there were rumours that it involved logic programming.

## Concurrent Prolog

One day in summer 1982 I received an unexpected call from Bob Kowalski. Udi Shapiro, who was well known for his work on algorithmic debugging, was visiting IC that day and was interested in the Relational Language. This took me by surprise because nobody had expressed an interest in our language before. Unfortunately, I had missed his visit.

The next thing I remember, probably in November 1982, was another rumour: Shapiro was visiting ICOT and had designed a language like ours; moreover, it was to be adopted by ICOT for the FGCS project. It was hard to be sure what this meant at first, but in February 1983 I received a copy of the Concurrent Prolog paper [12]. It was a brilliant paper, and is still one of my all-time favourites. The language was simple: guarded clauses, like the Relational Language, plus a read-only annotation for variables. There were a lot of examples and a small interpreter in Prolog. This was an excellent way to let people try the language for themselves, at a time when the usual way to distribute software was to mail tapes or disks in various nonstandard formats. The Concurrent Prolog interpreter was small enough for anyone to type in to a Prolog system and run.

I read the paper avidly. What I wanted to know was: could Concurrent Prolog do anything that the Relational Language couldn't? The answer was yes: *incomplete messages*. If a producer goal left unbound variables in an incrementally generated data structure, those variables could carry data from the consumer back to the producer or even between consumers. This seemed to be a powerful technique, but it wasn't possible in the Relational Language because of its strong modes: a producer might leave a variable unbound, but nobody else was allowed to bind it.

In fact, I had already found strong modes to be rather restrictive while writing the Relational Language compiler, in both Prolog and the Relational Language itself. I never needed backtracking but I did miss Prolog's "logical variable". But I thought it was a price worth paying: if we gave up strong modes, the language would no longer be implementable in a distributed way *and* our elegant and simple mode analysis would be messed up.

## Parlog 1983

The effect of Concurrent Prolog was dramatic. We decided to abandon strong modes, despite the disadvantages. Anyway, only a minor change to our language was needed: we added some annotations to override the default modes and changed the mode analysis algorithm accordingly. We hastily wrote a paper [2] about the latest version of the language and named it *Parlog*, for “PARallel LOGic”. The name *Paralog* had already been used by Tohru Moto-oka (one of the originators of the FGCS project) and others, but we didn’t realize then that Parlog was a common surname in Romania.

The Concurrent Prolog paper also made us rethink the implementation of the language. Although we already had a compiler for a state-of-the-art parallel architecture, Alice, that machine had yet to be built. It was hard for anyone, including us, to experiment with the language. We therefore suspended work on Alice implementation and instead quickly developed an implementation on top of Prolog.

## Parlog vs. Concurrent Prolog

1983-84 was a period of intense rivalry between Parlog and Concurrent Prolog. As outsiders, we didn’t know exactly what was going on at ICOT (later recounted in [8]), but it was obvious that the rumours were true: ICOT’s work was to be based on Concurrent Prolog. Naturally, we thought this was a bad idea, and this was partly for technical reasons: the language seemed quite expensive to implement. In general, it was impossible to prevent clause guards from binding variables in the calling goal, so a multiple environment mechanism was needed. Our method, where input and output arguments are distinguished, seemed to us inherently more efficient.

This was also a time when the amount of interest in logic programming, and especially concurrent logic programming, was increasing rapidly: a kind of reflected glory from the multi-billion yen FGCS project. Without FGCS, I have no doubt that the field would have remained an obscure backwater.

In October 1983, I was invited to ICOT for the first time, along with Keith and Udi Shapiro. ICOT was a wonderful place: a high-tech, high-rise lab overlooking quiet traditional streets. The ICOT people were extremely hospitable, and enthusiastic about concurrent logic programming. We were told about the proposed KL1 language and how Concurrent Prolog fitted into it, and had detailed discussions with Koichi Furukawa, Akikazu Takeuchi, Takashi Chikayama, and Kazunori Ueda, among others.

## Parlog 1984

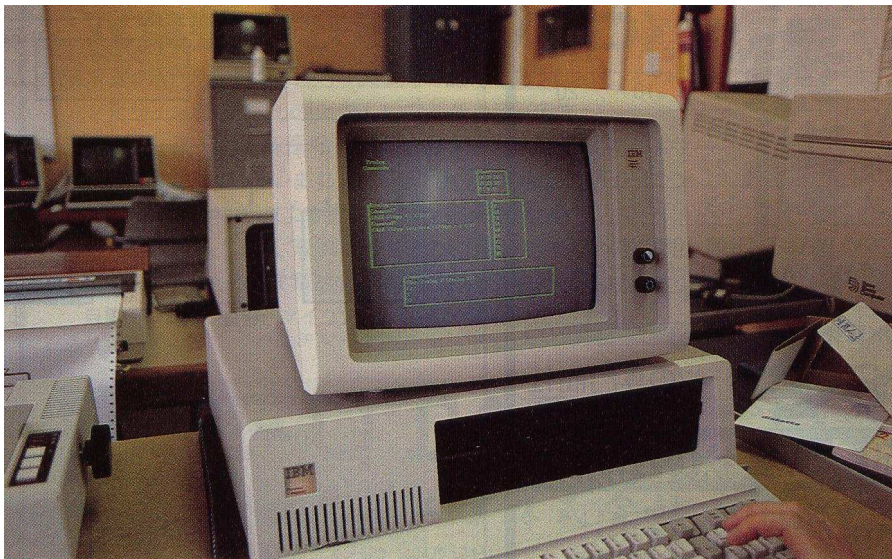
The first version of Parlog [2] was frankly a mess. Since the Relational Language, the language had somehow become complex and bloated. As well as the set constructors and functions, the annotations — needed for the improved mode analysis — made the language seem more complicated than it was. More seriously, changing to weak modes had caused a potential problem with guards: it was now possible for a guard to bind a variable in the calling goal.

We decided to give up functions because they were only easy to implement on the Alice machine. We were also able to get rid of all annotations just by restricting every predicate to a single mode. As in Prolog, the ability to use a predicate in more

than one mode is an interesting curiosity but rarely practical. These changes made the language a lot simpler, but there were some critics who disliked *any* mode declarations. To answer them, we showed how modes could be removed completely by compiling them to a “kernel” form, with specialized matching and testing goals in the guards and assignment goals in the bodies. All arguments could now be input arguments and mode declarations were just a syntactic convenience, not an essential part of the language.

To solve the problem with guards, we defined the concept of *safe guards*: input argument variables were now allowed in guards only in strong input positions. This guard safety property could be checked (conservatively) at compile time.

The design of this improved version of Parlog was completed by April 1984 [3], and a new implementation on top of Prolog followed.



Demonstrating *micro-Parlog* to a Japanese journalist in 1984.  
This Parlog implementation was later developed into *Parlog for Windows*,  
which is still available at <http://www.parlog.com/parlog>

## A turning point

I next visited Japan in early November 1984 to attend the second FGCS conference. This conference was a very big event: among the many foreigners attending, researchers seemed to be outnumbered by reporters, senior managers, and civil servants. Just after the conference, a high-level working lunch was arranged between representatives of MITI and the Alvey programme, the UK’s answer to FGCS. Mike Reeve and I — incongruously scruffy — were conscripted by a desperate Brian Oakley (director of Alvey) to make up the numbers on the “UK side”, since many of the suit-wearing bureaucrats had left on the Friday afternoon.

During the conference, in the evenings, we had several meetings with the ICOT researchers to discuss their latest findings. It seemed that they were having problems with Concurrent Prolog and were not happy with their progress. By the end of that trip, I had the impression, for the first time, that our approach was being taken seriously.

A bit later, in December 1984, I received a draft document [13] from Ueda listing in great detail some serious issues that he had identified with the Concurrent Prolog language definition. This was the result of many months' implementation work by several ICOT teams. About the same time, he sent me a draft of his proposal for a new language, later named Guarded Horn Clauses (GHC) [14], which he thought would avoid these problems. ICOT decided to give up Concurrent Prolog and switch to GHC.

## GHC

GHC was remarkably similar to Parlog. More precisely, it was like the kernel form of Parlog that we had introduced a few months earlier. The difference was that Ueda had cleverly unified the two concepts of guard suspension and guard safety. In GHC, a guard suspends if it tries to bind a variable in the calling goal.

Deciding which variables are goal variables could be expensive in general, but is very easy if the language is restricted to "flat" guards. So ICOT ended up implementing Flat GHC, which is effectively identical to Flat Parlog.

## Epilog

After 1984, the languages had stabilized and implementation efforts started in earnest. The size of the Parlog group (which was previously 1-2) increased and ICOT expanded even further. Some excellent, seriously usable, implementations of both Parlog and GHC were developed, especially Jim Crammond's Parallel Parlog system and ICOT's KLIC. Ironically, by the time that these implementations became available, there were few people to use them: interest in concurrent logic programming, and FGCS technology in general, had already started to fade.

## Lessons

An article like this wouldn't be complete without a list of "lessons learned" from the experience, so here are a few conclusions about language design:

1. ***Language design is difficult.*** Unlike most engineering endeavours, the things that make a programming language good or bad are hard to measure objectively.
2. ***Implement early.*** Many mistakes were made in the design of all early concurrent logic programming languages, which were obvious with hindsight. Practical experience with a language is a good substitute for hindsight.
3. ***Presentation is important.*** Chikayama [8] speculated that Concurrent Prolog may have been more popular at ICOT than the Relational Language because of its Prolog-like syntax. (In fact, at the time the Relational Language was conceived, there was no standard syntax for Prolog, but the Edinburgh syntax was becoming a *de facto* standard by the beginning of the ICOT era.)
4. To quote the words of one of the great FGCS researchers (which he wrote at the end of *every* email): ***Keep It Simple, Stupid!***

## Features of main concurrent logic programming languages

	Relational Language	Concurrent Prolog	Parlog 1983	Parlog 1984	GHC
Reference	[1]	[12]	[2]	[3]	[14]
Paper date	03/1981	02/1983	05/1983	04/1984	06/1985
Syntax: if	<-	:-	:-	<-	:-
commit				:	
concurrent and	//	,	,	,	,
sequential and	&		&	&	
conc search	.	.	.	.	.
seq search		otherwise	;	;	otherwise
atom var list	A x x.Nil	a X [X]	A x [x]	A x [x]	a X [X]
Incremental binding	List	Any term	Any term	Any term	Any term
Incomplete messages?	No	Yes	Yes	Yes	Yes
Suspension 1	Modes: input matching	Unifying read-only variable	Modes: input matching	Modes: input matching and testing in guard	Unifying goal variable in guard
Suspension 2	Modes: output assignment		Modes: output assignment	Guard goal suspension	Guard goal suspension
Output	Assignment	Unification	Assignment	Assignment	Unification
Number of modes	Many		Many	One	
Guards	Ground	Unrestricted	Not specified	Safe: compile-time	Safe: run-time
Multiple environments?	No	Yes	No	No	No
Communication	Asynchronous / synchronous / bounded	Asynchronous	Asynchronous / bounded	Asynchronous	Asynchronous
Extra feature 1			Set constructors	Set constructors	
Extra feature 2			Functions	Metacalls	

## References

1. Clark, K. L. and Gregory, S. 1981. A relational language for parallel programming. In *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture* (Portsmouth, New Hampshire, October 18-22, 1981), ACM Press, New York, 171-178. <http://doi.acm.org/10.1145/800223.806776>
2. Clark, K. L. and Gregory, S. 1983. PARLOG: a parallel logic programming language. Research Report DOC 83/5, Imperial College, London. <http://www.cs.bris.ac.uk/~steve/papers/doc83-5.pdf>
3. Clark, K. and Gregory, S. 1986. PARLOG: parallel programming in logic. *ACM Trans. Program. Lang. Syst.* 8, 1 (Jan. 1986), 1-49. <http://doi.acm.org/10.1145/5001.5390>
4. Clark, K.L., McCabe, F.G., and Gregory, S. 1982. IC-Prolog language features. In Clark, K.L. and Tarnlund, S.A. (Eds.), *Logic Programming*, Academic Press, London, 253-266.
5. van Emden, M.H. and de Lucena, G.J. 1982. Predicate logic as a language for parallel programming. In Clark, K.L. and Tarnlund, S.A. (Eds.), *Logic Programming*, Academic Press, London, 189-198.
6. Friedman, D.P. and Wise, D.S. 1976. CONS should not evaluate its arguments. In *Proc. 3rd Intl. Colloquium on Automata, Languages and Programming*, Edinburgh University Press, 257-284.
7. Friedman, D.P. and Wise, D.S. 1980. An indeterminate constructor for applicative programming. In *Proc. 7th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (Las Vegas, Nevada, January 28-30, 1980), ACM Press, New York, 245-250. <http://doi.acm.org/10.1145/567446.567470>
8. Fuchi, K., Kowalski, R., Furukawa, K., Ueda, K., Kahn, K., Chikayama, T., and Tick, E. 1993. Launching the new era. *Commun. ACM* 36, 3 (Mar. 1993), 49-100. <http://doi.acm.org/10.1145/153520.153541>
9. Gregory, S. 1980. Towards the compilation of annotated logic programs. Research Report DOC 80/16, Imperial College, London. <http://www.cs.bris.ac.uk/~steve/papers/doc80-16.pdf>
10. Hoare, C.A.R. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (Aug. 1978), 666-677. <http://doi.acm.org/10.1145/359576.359585>
11. Kahn, G. and MacQueen, D.B. 1977. Coroutines and networks of parallel processes. In *Proc. IFIP Congress 77*, North Holland, Amsterdam, 993-998.
12. Shapiro, E. 1983. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, Tokyo. <http://www.icot.or.jp/cgi-bin/tread.sh?TRNAME=tr0003>
13. Ueda, K. 1985. Concurrent Prolog re-examined. Technical report TR-102, ICOT, Tokyo. <http://www.icot.or.jp/cgi-bin/tread.sh?TRNAME=tr0102>
14. Ueda, K. 1986. Guarded Horn Clauses. In *Proc. 4th Conf. on Logic Programming '85* (Tokyo), Springer-Verlag, 168-179. [http://dx.doi.org/10.1007/3-540-16479-0\\_17](http://dx.doi.org/10.1007/3-540-16479-0_17)