# Logic Programming and Concurrency: A Personal Perspective

Kazunori Ueda
Department of Computer Science, Waseda University

May 7, 2006

I was asked to contribute a personal, historical perspective of logic programming (and presumably its relation to concurrency). I once wrote my personal perspective for Communications of the ACM in 1993 [1]. The article was also intended to record a detailed, historical account of the design process of Guarded Horn Clauses (GHC) and the kernel language (KL1) of the Japanese Fifth Generation Computer Systems (FGCS) project based on GHC. The readers are invited to read the CACM article, which is full of facts and inside stories. My view of the field remains basically the same after thirteen years.

Another related article appeared in the "25-Year Perspective" book on logic programming [2], in which I tried to convey the essence of concurrent logic/constraint programming and its connection to logic programming in general. If your view of concurrent logic/constraint programming is approximately "an incomplete variant of logic programming," I would like to invite you to read [2] and update the view. In a word, it is a simple and powerful formalism of concurrency.

In this article, I'll try to convey the essence of the field to the present audience, highlighting my beliefs, principles and lessons learned. I'll also briefly describe diverse offspring of concurrent logic/constraint programming.

## 1 The First Work on Logic Programming

Since I was a master student in late 1970's, I was interested in all kinds of programming languages, from practical to pedagogical to exotic. I met Prolog in this period but it took some while to be able to appreciate it (as is the case for most of the students we teach today) since it had a number of distinctive features.

My first journal paper (1983, with Hideyuki Nakashima and Satoru Tomura, in Japanese) was on declarative I/O and string manipulation in logic programming. Our motivations and beliefs behind the work were:

(i) *Declarative programming languages should not resort to side effects.*

(ii) *I/O is the central purpose of any program and programming language connected to the real world.*

These two principles, acquired before I joined the Japanese Fifth Generation Computer Systems (FGCS) project in 1983, formed the baseline of my work on language design for more than 20 years.

The second belief is not necessarily popular in the declarative language community, but I was very encouraged by reading Tony Hoare's paper on CSP (CACM, August 1978 issue), whose abstract started with:

> "This paper suggest that input and output are basic primitives of programming."

## 2 Development of Guarded Horn Clauses

The first three years of my FGCS project days were devoted to the design of the Kernel Language version 1 (KL1) of the project, which was about to start with the working hypothesis that KL1 should be based on Concurrent Prolog, a logic programming language with two new additional constructs, read-only annotation to express synchronization and the commit operator to express choice (don't-care) nondeterminism. The hypothesis, largely due to Koichi Furukawa and Akikazu Takeuchi, seemed bold to almost everybody inside the project including myself because the other form of nondeterminism, search, was absent there. Nevertheless, I soon believed this approach way was worth exploring. Ehud Shapiro convinced me of

(iii) *Occam's razor (or minimalism) in programming language design (by 1983, I learned two clear instances of this principle, CSP/Occam and Concurrent Prolog).*

In addition, I felt that

(iv) *different concerns (e.g., concurrency and search in our context) should be separated to understand things analytically; only after that they could be integrated (if possible at all).*

As soon as Concurrent Prolog was adopted as the working hypothesis, we launched an implementation project.

(v) *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.*

Three Concurrent Prolog interpreters, each differing in the mechanisms for binding environments, were almost complete by August 1984. However, we found several subtleties in the language, and felt that we should re-examine the language specification of Concurrent Prolog in depth before we went any further.

(vi) *In designing programming languages or calculi, giving a formal (or at least rigorous) definition is a means and not a purpose. The important next step is to scrutinize various properties and consequences of the definition.*

The purpose of the re-examination of Concurrent Prolog was on the granularity of two primitive operations, namely (1) unification under read-only annotations and (2) committed-choice after OR-parallel execution of multiple clauses. Although the way programming languages are defined has changed with the advent of Plotkin's SOS (structural operational semantics, also known as small-step semantics as opposed to late Gilles Kahn's natural (big-step) semantics), a lesson learned in this work is still valid:

(vii) *The small-step semantics of a language construct does not necessarily express the real atomic operations of the construct.*

To appreciate the above, it should be best to consider the basic operation of logic programming, resolution. A resolution step involves unification of arbitrarily large terms that can be performed in many smaller steps. You may also recall that, in chemistry and physics, both atoms (that once were considered indivisible) and subatomic particles are important, and in computer science the same principle applies to constructs of programming languages, especially those of parallel or concurrent languages.

The conception of Guarded Horn Clauses (GHC, December 1984) was an outcome of such subatomic consideration of parallel execution of logic programs. I suppose such an approach is not common even in the design of various concurrency formalisms, but the strength and the stability of GHC as a formalism comes exactly from here.

## 3 Logic Programming versus Concurrency

GHC and other concurrent logic languages were destined to wear two different hats: It was born from logic programming but its principal connection was to concurrency. At first, I used to describe GHC by comparing it with logic programming. However, it did not take too long until I viewed GHC as a simple model of concurrency that featured (or more precisely, allowed very simple encoding of) messaging, dynamic process creation, and reconfigurable channels.

(viii) *Concurrent logic programming is a model of concurrency featuring channel mobility in the sense of the pi-calculus (and channel fusion in the sense of the fusion calculus).*

And it was not just a model. It was a family of languages implemented and used by a number of research groups around the world. I think this is an important contribution of the logic programming community to the field of concurrency, which is not to be missed by ourselves.

At that time (1985–1986), the idea of constraint logic programming was not available to us, and "bindings", the algebraic counterpart of constraints, was the standard term used to speak of computational aspects of logic programs. Except for the terminology, however, we were already viewing computation along the *ask* and *tell* setting of concurrent constraint programming:

> "...it is quite natural to view a GHC program in terms of binding information and the agents that observe and generate it." "In general, a goal can be viewed as a process that observes input bindings and generates output bindings according to them. Observation and generation of bindings are the basis of computation and communication in our model." — [4]

Study of the foundations of concurrent logic languages came after practice, but time became ripe for concurrent logic programming to embrace ideas from constraint logic programming, leading to an elegant reformulation of the framework by Michael Maher and Vijay Saraswat. Concurrent constraint programming (CCP) became a more popular term of the framework.

However, terminology needs care and attention. One thing to be noted is:

(ix) *Concurrent logic languages—or at least some of them including GHC—are (instances of) concurrent constraint languages.*

I have found this fact very often forgotten in the literature, perhaps because it may sound like saying that an instance of a class was available before a class was defined. However, everybody will agree that Socrates was a mammal even if the notion of mammals was not available in his time.

I'm still much less confident in the names of the paradigm than its technical content. Any name would be fine when we talk to ourselves, but we must reach out to a broader community. Concurrent logic programming sounds like a branch of logic programming, which is true, but it is about concurrency as well. Likewise, concurrent constraint programming sounds like a programming paradigm, which is true, but it is a process calculus as well. Such a broad coverage is not easy to express in a single term, but let me propose the term

(x) *Constraint-Based Concurrency*

to mention the paradigm [3], as opposed to other process calculi in which names play principal roles and could be called *Name-based Concurrency*.

Let me get back to the connection between logic programming and concurrent logic programming. One thing to be stressed is:

(xi) *Concurrent logic programming may look remote from the rest of logic programming, but from the technical point of view it is not the case.*

As mentioned earlier, the design of GHC was an outcome of subatomic analysis of logic programming. We tried to retain the properties of logic programming wherever possible to establish a robust, rule-based formalisms for communicating programs. This was achieved by exploiting the beautiful properties of constraints and logical variables.

One instance of the previous claim, which I'd like the readers to recall, is:

(xii) *Dataflow synchronization a useful construct in logic programming with and without choice nondeterminism alike.*

Examples supporting this claims include delaying, coroutining, sound negation-as-failure, and the Andorra principle (due to D. H. D. Warren).

In conclusion, I would suggest that concurrent logic programming (and constraint-based concurrency in general) be viewed *not* as

(xiii, avoid this)
$$\begin{aligned}\text{Concurrent LP} &= \text{LP} + \text{committed choice}\\ &= \text{LP} - \text{completeness}\end{aligned}$$

but as [2]:

(xiv)
$$\begin{aligned}\text{Concurrent LP} &= \text{LP} + \text{directionality of dataflow}\\ &= \text{LP} + \text{embedded concurrency control.}\end{aligned}$$

The connection between constraint-based concurrency and name-based concurrency (as represented by the pi-calculus) is not well understood in the community, either. One of the purposes of my invited talk at TACS'01 [3] and the tutorial at ICLP'01 was to relate these formalisms.

# 4 Type Systems

The above view naturally motivates the study of program analysis and type systems that deal dataflow and directionality. Traditionally, the LP community has used the term "modes", but I found the term "types" (interpreted in a broad sense) much more appealing to the rest of the CS community, hence the slogan:

(xv) *Modes are types in a broad sense.*

In other words, mode systems are a kind of non-standard type systems whose domain are more than sets of (first or higher-order) values.

Static analysis of dataflow may sound contrary to the spirit of constraint programming since constraint propagation is supposed to take place in arbitrary directions. However, constraint-based concurrency turned out to be an interesting place where the power of constraint programming (powerful enough to encode first-class mobile channels) and the power of static type systems meet:

(xvi) *Constraint programming involves complicated dataflow, but there are cases where the complicated dataflow can be statically analyzed.*

Thus during 1990's I worked on type systems for constraint-based concurrency that dealt with modes and linearity (the number of readers of each variable) as well as values. Kima, a tool (still available on the web) for type-based program diagnosis and fully automatic debugging was based on the mode system of Moded Flat GHC. An important general lesson I learned from that work was:

(xvii) *A type system can tell much more than whether a program is well-typed or ill-typed.*

This observation is thanks to the constraint-based (as opposed to proof-theory-style) formulation of our type system. To make the above lesson more specific, I learned:

(xviii) *Constraint-based program analysis, if done in an appropriate setting, can pinpoint program errors and even correct them.*

## 5 The Paradigm's Diverse Offspring

Constraint-based concurrency yielded diverse offspring, all with unique features. The most successful offspring up to now should be Constraint Handling Rules (CHR), which addressed constraint programming, the field in which a high-level programming language was called for. I did not foresee that an extension of concurrent logic programming languages has made such a success in a quite different field.

Another important offspring is timed and hybrid concurrent constraint programming, which gave a high-level programming language with rich control structure to the field in which (a timed or hybrid version of) automata were the basic means of problem description.

My own recent work is the design and implementation of LMNtal (pronounced "elemental"), a model and a full-fledged language for hierarchical graph rewriting. This is another field whose programming language aspects have been scarcely studied. LMNtal was an outcome of the attempt to unify constraint-based concurrency and CHR.

LMNtal is a lean language in the sense that it does not even feature constant symbols and its variables are used only for indicating one-to-one connection between atomic formulae. Nevertheless, it turned out to be a very versatile language related to many formalisms including various process calculi, multiset rewriting, the lambda calculus, and (of course) logic programming. The main feature of LMNtal is its ability to deal with connectivity and hierarchy simultaneously in a simple setting.

(xix) *Connectivity and hierarchy are the two structuring mechanisms found in many fields ranging from society to biology, not to mention the world of computing.*

Another formalism motivated by this observatin is Robin Milner's Bigraphical Reactive Systems.

There may be more connections from constraint-based concurrency to emerging technologies, but the above instances should be sufficient to support my final remark, which I hope to share with the readers for the development of the field:

(xx) *Logic programming today embraces diverse interesting technologies beyond computational logic (in a narrow sense) as well as those within computational logic.*

## References


[1] Shapiro, E. Y., Warren, D. H. D., Fuchi, K., Kowalski, R. A., Furukawa, K., Ueda, K., Kahn, K. M., Chikayama, T. and Tick, E., The Fifth Generation Project: Personal Perspectives. *Comm. ACM*, Vol. 36, No. 3 (1993), pp. 46–103.
(This is actually a collection of single-authored articles, and my article (pp. 65–76) was originally titled "Kernel Language in the Fifth Generation Computer Project.")

[2] Ueda, K., Concurrent Logic/Constraint Programming: The Next 10 Years. In *The Logic Programming Paradigm: A 25-Year Perspective*, Apt, K. R., Marek, V. W., Truszczynski M., and Warren D. S. (eds.), Springer-Verlag, 1999, pp. 53–71.
(Extensive references to related work and technologies can be found in this paper.)

[3] Ueda, K., Resource-Passing Concurrent Programming. In *Proc. Fourth Int. Symp. on Theoretical Aspects of Computer Software (TACS'01)*, Lecture Notes in Computer Science 2215, Springer, 2001, pp.95–126.

[4] Ueda, K., Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. ICOT Tech. Report TR-208, ICOT, Tokyo, 1986. Also in *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, Amsterdam, 1988, pp. 441–456.