Concurrent Logic/Constraint Programming: The Next 10 Years *

Kazunori Ueda

Department of Information and Computer Science
Waseda University
4-1, Okubo 3-chome, Shinjuku-ku, Tokyo 169-8555, Japan
ueda@ueda.info.waseda.ac.jp

Abstract. Concurrent logic/constraint programming is a simple and elegant formalism of concurrency that can potentially address a lot of important future applications including parallel, distributed, and intelligent systems. Its basic concept has been extremely stable and has allowed efficient implementations. However, its uniqueness makes this paradigm rather difficult to appreciate. Many people consider concurrent logic/constraint programming to have rather little to do with the rest of logic programming. There is certainly a fundamental difference in the view of computation, but careful study of the differences will lead to the understanding and the enhancing of the whole logic programming paradigm by an *analytic approach*. As a model of concurrency, concurrent logic/constraint programming has its own challenges to share with other formalisms of concurrency as well. They are: (1) a counterpart of λ-calculus in the field of concurrency, (2) a common platform for various non-sequential forms of computing, and (3) type systems that cover both logical and physical aspects of computation.

1 Grand Challenges

It seems that concurrent logic programming and its generalization, concurrent constraint programming, are subfields of logic programming that are quite different from the other subfields and hence can confuse people both inside and outside the logic programming community.

While most subfields of logic programming are related to artificial intelligence in some way or other—agents, learning, constraints, knowledge bases, automated deduction, and so on—, concurrent logic/constraint programming is somewhat special in the sense that its principal connection is to concurrency.

Concurrency is a ubiquitous phenomenon both inside and outside computer systems, a phenomenon observed wherever there is more than one entity that

* To appear in *The Logic Programming Paradigm: A 25-Year Perspective*, K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren (eds.), Springer-Verlag, 1999, pp. 53–71.

may interact with each other. It is important for many reasons. Firstly, the phenomenon is so ubiquitous that we need a good theoretical and practical framework to deal with it. Secondly, it is concerned with the infrastructure of computing (the environment in which computer systems and programs interact with the rest of the world) as well as activities within computer systems, and as such the framework scales up. In other words, it is concerned with *computing in the large*, and accordingly, programming in the large. Thirdly, it encompasses various important forms of non-sequential computing including parallel, distributed, and mobile computing.

Bearing this in mind, I'd like to propose the Grand Challenges of concurrent logic/constraint programming.[1] The theoretical computer science community is struggling to find killer applications, but I would claim that, as far as concurrency is concerned, there are at least three important scientific challenges besides finding killer applications.

1. *A "λ-calculus" in the field of concurrency.* It is a real grand challenge to try to have a model of concurrency and communication which is as stable as λ-calculus for sequential computation.
   We have had many proposals of models of concurrency: Petri Nets [36], Actors [2], Communicating Sequential Processes [24], and many formalisms named X-calculus, X being Communicating Systems [29], π [30], Action [31], Join [17], Gamma [4], Ambient [6], and so on.
   Concurrent constraint programming [38] is another important model of concurrency, though, unfortunately, it is often overlooked in the concurrency community. I proposed Guarded Horn Clauses (GHC) as a simple concurrent logic *language*, but in its first paper I also claimed:

   > "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."
   >
   > — [51] (1985)

   One of the reasons why there are so many models is that there are various useful patterns of interaction, some of which are useful for high-level concurrent programming and others rather primitive. Here, a natural question arises as to whether we can find a lowest possible layer for modeling concurrency. I am not sure if people can agree upon a single common substrate, but still believe that the effort to have a simple and primitive framework (or

[1] Concurrent constraint programming can be viewed both as a generalization of concurrent logic programming and as a generalization of constraint logic programming. This article will focus on the former view since the challenges of concurrent constraint programming from the latter view should be more or less similar to those of constraint logic programming.

a few of them) is very useful and will lead to higher respect of the field. Note that everybody respects λ-calculus but it still has a number of variants and some insist that it is not fully primitive (see, for example, [1]).
Needless to say, a stable calculus is a challenge but is not an ultimate goal. What we need next is a high-level programming language fully supported by a stable theory.

2. *Common platform for non-conventional computing.* The next challenge is to see if a common platform—the pair of a high-level concurrent language and an underlying theory—can be the base of various forms of non-conventional computing such as
   - parallel computing,
   - distributed/network computing,
   - real-time computing, and
   - mobile computing.

   Historically, they have been addressed by more or less different communities and cultures, but all these areas share the following property: unlike conventional sequential computing, programmers must be able to access and control the physical aspects of computation. At the same time, programmers don't want to be bothered by physical considerations in writing correct programs and porting them to different computing environments. These two requirements are referred to as *awareness* and *transparency* (of/from physical aspects).
   The fact that all these areas have to do with physical aspects means that they all have to do with concurrency. They all make sense in computing environments participated in by more than one physical entity such as 'sites' and 'devices'. This is why it is interesting to try to establish a novel unified platform for these diverse forms of non-conventional symbolic computing.

3. *Type systems and frameworks of analysis for both logical and physical properties.* The third grand challenge is a framework of static analysis to be built into concurrency frameworks. The first thing to be designed is a type system. Here I use the term "type system" in its broadest sense; that is, to have a type system means to:
   (a) design the notion of types, where the notion can be anything that is well-defined and useful either for programmers or for implementations,
   (b) define typing rules that connect the world of program text and the world of types, and
   (c) establish basic (and desirable) properties of well-typed programs such as subject reduction and strong normalization.

   So a type does not necessarily represent a set of possible values a syntactic construct can denote in the standard semantics; for instance, a mode (directionality of information flow) is thought of as a type in a broad sense. As we know, types play extremely important roles in programming languages and calculi. The fundamental difference between types and other formalisms of program analysis (such as abstract interpretation) is that, although well-typedness imposes certain constraints on allowable programs, the notion of types is exposed to programmers. Accordingly, types should be accessible to programmers and should help them understand and debug their programs better.

These features of type systems are expected to play key roles in concurrent programming. A challenge here is to deal with physical as well as logical properties of programs in a way accessible to programmers.

I believe addressing these scientific challenges is as essential as building killer applications because, only with such endeavor, declarative languages and theory-driven approach can find their raisons d'êtle.

2 Two Approaches to Addressing Novel Applications

It is natural to think that addressing novel applications requires a powerful programming language with various features. A popular approach to making a logic programming language more powerful is to generalize it or to integrate useful features into it. Constraint logic programming, inductive logic programming, higher-order logic programming, disjunctive logic programming, etc. are all such generalizations. Some extensions are better thought of as integration rather than generalization; examples are functional logic programming and multi-paradigm extensions such as Oz [46].

However, there is a totally different approach to a more powerful language, which I call an *analytic approach.* In an analytic approach, one tries to identify smaller fragments of logic programs (or of extensions of logic programs) with nice and useful properties that may lead to efficient implementation.

Note that what I mean by "powerful" here is not in terms of expressive power. By identifying possibly important fragments of a general framework and studying them carefully, one may able to establish new concepts with which one can understand the whole framework in more depth and detail. Also, one may find that some fragment allows far more efficient implementation. (A popular example where simplicity is the source of efficiency is the RISC architecture.) One may build programming tools that take advantage of the properties of fragments. They are a source of power because it may open up new application areas that could not be addressed by the general framework.

The above claim could be understood also from the following analogy: having a notion of Turing machines does not necessarily mean that we don't have to study pushdown or finite-state automata. They have their values in their own rights. Another example is the relationship between untyped and typed λ-calculi. Yet another obvious example is the identification of Horn sentences from full first-order formulae, without which the logic programming paradigm would not exist today. Examples of smaller fragments of logic programming languages that have been studied in depth are Datalog (no function symbols) and concurrent logic languages (no search in exchange of reactiveness).

The analytic approach is useful also when one attempts to generalize or integrate features. Integration will succeed only after the features to be integrated have been well understood and the interface between them has been carefully designed. A criterion of success is whether one can give clean semantics to the whole integrated framework as well as to each component. If the components interact only at the meta (or extralogical) level, the whole framework is considerably more complicated (in terms of semantics) than their components, which means the verification and manipulation of programs become considerably harder. This issue will be discussed in the next section.

3 Logic Programming vs. Concurrent Logic Programming

Concurrent logic programming was born from the study of concurrent execution of logic programs. It turned out to enjoy a number of nice properties both as a formalism and as a language for describing concurrency. In the logic programming community, however, concurrent logic programming has always been a source of controversy. Unfortunately, the controversy was by and large not very technical and did not lead to deeper understanding of the paradigms.

A typical view of concurrent logic programming has been:

$$\begin{aligned}\text{Concurrent LP} &= \text{LP} + \text{committed choice}\\ &= \text{LP} - \text{completeness}\end{aligned}$$

Although both the first and the second equations are not totally wrong, viewing committed choice simply as losing completeness is too superficial.

Committed choice or don't-care nondeterminism is an essential construct in modeling reactive computing and has been studied in depth in the concurrency community. It is essential because one must be able to model a process or an agent that performs *arbitration*. (For instance, a receptionist will serve whoever *(s)he thinks* comes first, rather than whoever comes first.) Semantically, it is much more than just discarding all but one of possible execution branches. All the subtleties lie in what events or information should be the basis of choice operations; in other words, the subtleties lie in the semantics of guard rather than the choice itself. The presence or absence of nondeterminism makes fundamental difference to the denotational semantics of concurrency (see [24] for example). When I was designing Guarded Horn Clauses, I believed don't-care nondeterminism was so essential that it was a bad idea to retain both don't-care nondeterminism and don't-know nondeterminism in a single model of computation.

Nevertheless, we also found in our experiences with concurrent logic languages that most of the predicates are deterministic and very few predicates perform arbitration—even though they change the whole semantical framework. Thus the aspect of arbitration is not to be overstated. A much more productive view of concurrent logic programming will accordingly be:

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

This is more productive because it emphasizes the aspect of dataflow synchronization, an important construct also in logic programming without committed choice. Examples where dataflow synchronization plays important roles include delaying, coroutining, sound negation-as-failure, and the Andorra principle [37]. Dataflow-centered view of the execution logic programs best captures the essence of concurrent logic/constraint programs, as became clear from the $ask + tell$ formulation advocated by Saraswat [38].

Another reason why the above view is more productive is that it addresses mode systems that prescribe the directionality of dataflow. Mode systems are attracting more interest in various subfields of logic programming because

– it shares with type systems many good properties from which both programmers and implementations can benefit, and
– many (if not all) predicates we write have a single intended mode of use, and there are a lot of situations where this fact can be exploited in interesting ways.

For example, Mercury [47] takes full advantage of strong moding to yield very efficient code.[2] Inductive logic programming benefits from moding in reducing search space [32]. Concurrent logic/constraint programming benefits enormously from strong moding both in implementation and programming [61, 63, 13, 3], and I strongly believe that

$$\text{Moded Concurrent LP} = \text{ask} + \text{tell} + \text{strong moding}$$

is one of the most flexible realistic models of concurrency.

It is vital to see that ordinary logic programming and concurrent logic programming are targeted at different scopes. What logic programming is concerned with include knowledge representation, reasoning, search, etc., while concurrent logic programming aims at a simple programming and theoretical model of concurrency and communication. Accordingly, concurrent logic languages should aim at general-purpose algorithmic languages which can potentially act as coordinators of more application-specific logic languages.

Since the conception of concurrent logic programming, how to reconcile two essential features in parallel knowledge information systems, search and reactiveness, has been one of the most difficult problems. The two paradigms could be integrated but should be done with utmost care. The solution adopted in PARLOG [14] was to use all-solutions predicates (à la `findall` in Prolog) to interface between the world of don't-know nondeterminism and the world of don't-care nondeterminism. The key issue here is how to gather multiple solutions obtained from different binding environments into a single data structure. Whether the all-solutions construct can be given clean, declarative meaning and whether it allows efficient implementation depend on the program and the goal for which

[2] Note, however, that the mode system of Mercury is very different from the mode system of Moded Flat GHC discussed in this paper; the former deals with the change of instantiatedness, which is a temporal property, while the latter deals with polarity, which is a non-temporal property [63].

solutions are to be collected [33, 55]. Roughly speaking, the requirement has to do with the proper treatment of logical variables. Existing all-solutions predicates in Prolog involve the copying of solutions, exhibiting *impedance mismatch*.[3]

Moding seems to play an important role here; we conjecture that all-solutions predicates can be given simple, object-level semantics if the program and the goal are well-moded under an appropriate mode system similar to the mode system for Moded Flat GHC [61].

Another example where moding played an important role in essentially the same way is the First Order Compiler [41], a compiler from a class of full first-order formulae into definite clauses.

The issue of clean interfacing arises in constraint logic programming systems also. In realistic applications of constraint satisfaction, it is often crucial that a constraint solver can run concurrently with its caller so that the latter be able to observe and control the behavior of the former incrementally. However, language constructs for doing so are yet to be refined.

4 An Application Domain: Parallel/Network Programming

Where should concurrent logic/constraint programming languages find promising applications? I believe that the most important areas to address are parallel and network applications for a number of reasons:

1. Even "modern" languages like Obliq and Java feature rather classical concurrency constructs such as monitors and explicit locking. In more traditional languages like C or Fortran, parallel/network programming is achieved with APIs such as MPI, Unix sockets, and POSIX threads. These constructs are all low-level compared with synchronization based on dataflow and arbitration based on choice, and programming with APIs seems to be a step backwards from writing provably correct programs even though verification is not impossible.
2. Parallel computing and distributed computing are considerably more difficult than sequential computing. Good models and methodologies to build large applications quickly are desperately called for.
3. These areas are becoming increasingly popular in a strong trend towards large-scale global computing environments both for high-performance dis-

[3] Oz features *computation spaces* (the pair of a local constraint store and a set of processes working on the store) as first-class citizens, with which encapsulated search can be programmed as higher-order combinators [45]. This approach is certainly cleaner in that a set of solutions is represented using (procedures returning) computation spaces instead of copied terms. In an analytic approach, however, we are interested in identifying a class of programs and goals for which a set of solutions can be represented without using higher-order constructs.

tributed computing [16] and for virtual network communities such as Virtual Places, Community Places and Matrix.[4]

4. These areas give us a good opportunity to demonstrate the power of small and yet "usable" languages with an appropriate level of abstraction. It seems essential to keep the languages simple enough—and much simpler than Java—to be amenable to theoretical treatment and to make them as easy to learn as possible. I anticipate that amenability to theoretical treatment, if well exploited, will be of enormous practical importance in these areas.

From a language point of view, the last point is the most challenging. Consider writing secure network applications with mobile code. This involves various requirements:

– Specification of *physical locations* (sites) where computation should take place.
– Reasoning about *resources*, such as time, stack and heap, that the computation may consume. Without it, downloaded code might make a so-called DoS (denial of service) attack by monopolizing computation resources.
– Security at various levels. Some of the high-level properties such as consistency of communication protocols can be guaranteed by typing and moding. Other high-level security issues may require more sophisticated analysis and verification. Low-level security could partly be left to Java's bytecode verifier if we use Java or Java bytecode as a target language.
– Transmission of various entities across possibly heterogeneous platforms. In addition to program code, linked data structures and symbols (usually given unique IDs locally on each site) are the main points of consideration in symbolic languages.

The requirements are so complicated and diverse that addressing them in an ad hoc way would result in a theoretically intractable language. It is an interesting and big challenge to obtain a language that allows and encourages formal reasoning about physical and logical properties of programs.

It may be a good idea for declarative language communities to share a set of (more concrete) challenges of the form "how can we program X in our formalisms?" to facilitate comparison between different paradigms. Instances of X may be:

– dynamic data structures (e.g., cyclic graphs; most declarative languages just ignore them, regrettably),
– live access counters of WWW pages,
– teleconferencing, and
– MUD (multi-user dungeon; text-based virtual reality).

[4] Interestingly, the designers of Grid [16], Virtual Places, Community Places and Matrix have all worked actively on concurrent logic/constraint programming.

5 Experiences with Guarded Horn Clauses and KL1

Although the progress has been admittedly slow, I am quite optimistic about the future of concurrent logic/constraint programming. My optimism is based on 15 years of our experiences with the paradigm since the initial stage of the Japanese Fifth Generation Computer Systems (FGCS) project. Figures 1–2 show the history of Guarded Horn Clauses (GHC) and KL1 as well as related events. The role and the history of the kernel language in the FGCS project are discussed in detail in my article in [44].

- 1983 Concurrent Prolog [42] and initial version of PARLOG [14]
- 1983-84 Big controversy (inside ICOT) on LP vs. concurrent LP for parallel knowledge information processing systems [44]
- 1985 First paper on GHC [51]
- 1985 GHC-to-Prolog compiler [52] used in our initial experiments
- 1985–86 GHC considered too general; subsetted to Flat GHC
- 1986 Prolog-to-GHC compiler performing exhaustive search [53]
- 1987 MRB (1-bit reference counting) scheme for Flat GHC [9]
- 1987 First parallel implementation of Flat GHC on Multi-PSI v1 (6 PEs) [25]
- 1987 ALPS [28] gave a logical interpretation of communication primitives
- 1987–1988 KL1 designed based on Flat GHC, the main extension being the *shoen* construct [57]
- 1988 Parallel implementation of KL1 on Multi-PSI v2 (64 PEs) [34]
- 1988 Strand [15] (evolved later into PCN [7] and CC++ [8])
- 1988 PIMOS operating system [10]
- 1988 Unfold/fold transformation and transaction-based semantics [54]
- 1989 Concurrent Constraint Programming [38]
- 1989 Controversy on atomic vs. eventual tell (Kahn's article in [44])
- 1989 MGTP (Model Generation Theorem Prover in KL1) project [19]

Fig. 1. GHC, KL1, and related events (Part I)

5.1 GHC as the Weakest Fragment of Concurrent Constraint Programming

After 13 years of research, heated discussions and programming experiences since the proposal of GHC, it turned out that this simplest fragment of concurrent constraint programming was surprisingly stable and versatile.

Let us see why it was so stable. GHC is thought of as the weakest Concurrent Constraint Language in the following senses: First, it features *ask* and *eventual tell* (i.e., publication of constraints *after* committed choice) but not *atomic tell* (publication *upon* committed choice). Second, its computation domain is a set of finite trees. Nevertheless, GHC as well as its ancestors featured

- 1989 Message-oriented implementation of Flat GHC [58]
- 1990 Mode systems for Flat GHC [58]
- 1990 Structural operational semantics for Flat GHC [59]
- 1990 Janus [39]
- 1991 Denotational semantics of CCP [40]
- 1991 AKL [26] (later evolved into Oz, Oz2, and Oz3)
- 1992 Parallel implementation of KL1 on PIM/m and PIM/p [48, 23]
- 1992 Various parallel applications written in KL1, including OS, biology, CAD, legal reasoning, automated deduction, etc. [11, 22, 35]
- 1992 Message-oriented parallel implementation of Moded Flat GHC [60]
- 1992 KLIC (KL1-to-C compiler) designed [12]
- 1992 MGTP solved an open problem (IJCAI'93 award) [20]
- 1994 Proof system for CCP [5]
- 1994 Moded Flat GHC formulated in detail [61]
- 1994 ToonTalk, a visual CCP language [27]
- 1995 Constraint-based mode systems put into practice [63]
- 1996 klint, a mode analyzer for KL1 programs
- 1996 Strong moding applied to constraint-based error diagnosis [13]
- 1997 kima, a diagnoser of ill-moded programs
- 1997 KLIEG, a visual version of KL1 and its programming environment [50]
- 1997 Strong moding applied to constraint-based error correction [3]

Fig. 2. GHC, KL1, and related events (Part II)

fundamental constructs for a concurrent programming language from the very beginning:

- parallel composition,
- creation of local variables,
- nondeterministic (committed) choice,
- value passing, and
- data structures (trees, lists, etc.).

The last two points are in contrast with other models of concurrency such as CCS and (theoretical) CSP that primarily focused on atomic *events*. GHC was proposed primarily as a concurrent language, though it was intended to be a model as well (Section 1).

Furthermore, GHC as well as its ancestors had the following features from the beginning:

1. *Reconfigurable process structures.* Concurrent logic languages supported dynamic reconfiguration of interprocess communication channels and dynamic creation of processes. Most mathematical models of concurrency, on the other hand, did not feature reconfigurable process structures until π-calculus was proposed in late 1980's.

2. *Object (process) identity.* This is represented by logical variables (occurring as the arguments of processes) through which processes interact with each other. Although objects themselves are not first-class in GHC, the variables identifying processes can be passed around to change the logical configuration of processes. Hence the processes were effectively mobile exactly in the sense of mobile processes in π-calculus.
3. *Input/output completely within the basic framework.* The input/output primitives of "declarative" languages had generally been provided as marginal constructs and in a quite unsatisfactory manner. I thought the design of general-purpose languages should proceed in the opposite way *by taking the semantics of input/output constructs as a boundary condition of language design.* Concurrent logic programs are often thought of as less declarative than logic programs, but real-life concurrent logic programs projected to logic programs (by forgetting synchronization) are much more declarative than real-life Prolog programs.

Later on, several important features were added:

1. KL1 [57] featured the notion of physical locations, though in a primitive form, to allow programmers to describe load balancing of parallel computation.
2. The mode system [61] introduced the notion of (statically decidable) *read/write capabilities* or *polarities* into each variable occurrence and each position of (possibly nested) data structures. In well-moded programs, a write capability can be passed around but cannot be copied or discarded, while a read capability can be copied and discarded. Well-modedness can be established by constraint-based mode analysis which is essentially a unification problem over feature graphs.
3. Linearity analysis, which distinguishes between one-to-one and one-to-many communication [64], enabled compile-time garbage collection and turned out to play a key role in parallel/distributed symbolic computation. For instance, parallel operations on an array in shared memory can be done without any interference by splitting the array into pieces in-place, letting parallel processes operate on its own piece in-place, and then merging the resulting pieces in-place [62]. Both mode analysis and and linearity analysis support resource-conscious programming by being sensitive to the number of occurrences of variables.

5.2 Logical Variables as Communication Channels

Most of the outstanding features of concurrent logic/constraint languages come from the power and the flexibility of logical variables as communication channels. Logical variables support:

– data- and demand-driven communication,
– messages with reply boxes,
– first-class channels (encoded as lists or difference lists),
– replicable read-only data, and

– implicit redirection across sites.

It is surprising that all these features are supported by a single mechanism. This uniformity gives tremendous benefits to theoretical foundations and programming systems, since a single framework can cover all these features.

5.3 Evolution as Devolution

It is generally understood that concurrent logic programming evolved into concurrent constraint programming by ALPS's logical interpretation of communication primitives [28] and Saraswat's reformulation of concurrent logic programming as a framework of concurrency [38]. However, I have an impression that the role of concurrent constraint programming as a generalization of concurrent logic programming has been a bit different from the role of constraint logic programming as a generalization of logic programming. While constraint logic programming has found several useful domains and applications, the main contribution of concurrent constraint programming has been in the understanding of the essence of the framework and the promotion of the study of semantics. The set of useful general-purpose constraint systems (other than obvious ones such as finite trees, integers and floating-point numbers) for concurrent constraint programming is yet to be identified.

Indeed, the history of the practice of concurrent logic programming could be summarized as "evolution by devolution" [49]. As conjectured in [51] (quoted in Section 1), GHC as the weakest fragment of concurrent constraint programming (d)evolved first by disallowing nested guards (Flat GHC) and then by featuring a mode system. Virtually all programs now written in KL1 and run by KLIC [12] are well-moded, though KLIC currently does not support mode analysis. On the other hand, there are only a few constructs added to KL1: *shoen* (a Japanese word meaning 'manor') as a unit of observing and controlling computation, the `@node( )` construct for process migration, and priorities.

Strong moding has degenerated unification (a case of constraint solving) to assignment to a variable, but has made GHC a much securer concurrent language; that is, it guarantees that constraints to be published to a shared store (binding environment) are always consistent with the current store. Linearity analysis guarantees that some class of programs (including most sorting programs, for instance) can run without generating garbage. Both analyses are useful not only for efficient implementation but also for the precise analysis of computational cost, which is essential in real-time computing and network programming with mobile code. In this way, degeneration may find new applications which could not be addressed by more general languages.

Oz [46, 21] has taken a totally different approach from GHC. It has incorporated a number of new constructs such as ports (a primitive for many-to-one communication), cells (containers of values that allow destructive update), computation space (encapsulated store, somewhat affected by nested guards of full GHC and KL1's *shoen*), higher-order, etc., and has moved from fine-grained to coarse-grained concurrency. It still encompasses a concurrent constraint language, but is now better viewed as a multi-paradigm language.

In contrast, I'd like to keep GHC a *pure* concurrent constraint language. (Moded Flat) GHC is quite a small fragment but it is yet to be seen what additional constructs are really necessary to make pure concurrent logic/constraint languages usable.

6 Some Failures and Problems

Although I'm optimistic about its future technically, concurrent logic/constraint programming has experienced a number of non-technical problems.

1. *Misleading names of the paradigms.* Concurrent logic languages are primarily *concurrent* programming languages though they retain the nice properties of logic programming wherever possible, such as soundness of proof procedures and declarative reading.[5] Unfortunately, concurrency is so unpopular in the logic programming community that concurrent logic programming often sounds like nothing more than an incomplete variant of logic programming. (An even worse name once used was *committed-choice languages.*)
   Concurrent constraint programming (languages) sounds better in this respect, but it has another problem. The conception of concurrent constraint programming is often said to date from ALPS, but this often results in the ignorance of its pre-history, the era of concurrent logic programming. *Concurrent logic languages are, by definition, (instances of) concurrent constraint languages.*[6]
2. *Community problem.* Although concurrent constraint programming is an elegant formalism of concurrency, it was born from logic programming, a paradigm quite unpopular in the concurrency community and the community of concurrent programming. So, this important paradigm can very easily be forgotten by both the logic programming community and the communities of concurrency theory and concurrent programming!

[5] One may argue that whether a concurrent language retains nice properties of logic programming is not very important, but this is not true. This criterion worked as a strong guideline of language design and resulted in many desirable properties as a concurrent language.

[6] At an early stage, I had understood GHC computation in terms of the exchange of bindings between processes rather than a restricted proof procedure:

> "... 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."
>
> — [56] (1986)

Of course, it was definitely after the proposal of concurrent constraint programming that binding- (constraint-) centered view became popular and the study of semantics made progress.

3. *Shortage of communication with neighboring communities.* This is a very general tendency and unfortunately applies to various communities related to concurrency. There are many techniques independently invented in the functional programming community, the (concurrent) object-oriented programming community and the (concurrent) logic programming community. Declarative arrays, frameworks of program analysis, scheduling of fine-grained tasks, and distributed memory management are all such examples.
   A bit more technical reason why concurrent logic/constraint programming is still unpopular in the concurrency community at large may be that its formulation looks rather indirect—popular and mundane idioms of concurrent programming such as objects, messages, and channels are all encoded entities.
4. *Few research groups.* Except for research groups on semantics, there are only two active (virtual) groups working on languages and implementation; one is the group working on Oz and the other working on GHC/KL1. Many key people who founded the field "graduated" too early before the paradigm became well understood and ready to find interesting applications. However, a good news is that most of them are working on the potential applications of the paradigm discussed earlier in this article. This leaves us a challenge to bridge the gap between what the paradigm offers and what the applications require.
5. *Textbooks.* Good textbooks and tutorial materials are yet to be published. There are some on concurrent logic programming, such as Shapiro's survey [43], but a tutorial introduction to the semantical foundations is still awaited. It's time to recast important concepts and results scattered over technical papers and re-present them from the current perspective.

7 Conclusions

Concurrent logic/constraint programming has been a simple and extremely stable formalism of concurrency, and at the same time it has been a full-fledged programming language. It is unique among many other proposals of concurrency formalisms in that information, communication and synchronization are modeled in terms of constraints, a general and mathematically well-supported framework. It is unique also in that its minimal framework (with *ask*, *eventual tell*, parallel composition, guarded choice and scoping) is almost ready for practical programming. That is, there is little gap between theory (computational model) and practice (programming language). The stability of the core concurrent constraint programming with *ask*+ *eventual tell* indicates that the logic programming community came up with something essential in early 1980's and noticed it in full in late 1980's.

There seems to be a feeling that concurrent logic/constraint programming has established an independent scientific discipline *outside* the logic programming paradigm. There is certainly a fundamental difference in how computation is viewed—one is on deduction while the other is on reactive agents. However, the fact that each paradigm features what the other does not have and the fact that they still share a lot of technicalities at less fundamental levels strongly indicate that they should benefit from each other. An interesting example of such bridging can be found in Constraint Handling Rules [18], a concurrent constraint language specifically designed for programming constraint systems.

Since concurrent constraint languages aim at general-purpose languages, they can benefit from static analysis more strongly than logic programming languages can. So I'd like to conclude this article by claiming that constraint-based static analysis can make concurrent constraint programming a simple, powerful, and safe language for

- parallel and high-performance computing,
- distributed and network computing, and
- real-time and mobile computing.

Its role in concurrent constraint programming is analogous to, but probably more than, the role of type systems in λ-calculus.

Acknowledgments

Comments and suggestions from anonymous referees were extremely useful in improving the paper. The title of the paper was adapted from the title of the Dagstuhl Seminar (No. 9741) on concurrent constraint programming held in October 1997.

References

1. Abadi, M., Cardelli, L., Curien, P.-L. and Lévy, J.-J., Explicit substitutions. *J. Functional Programming*, Vol. 1, No. 4 (1991), pp. 375–416.
2. Agha, G. A., *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, MA, 1986.
3. Ajiro, Y., Ueda, K. and Cho, K., Error-correcting Source Code. In *Proc. Fourth Int. Conf. on Principles and Practice of Constraint Programming (CP98)*, LNCS 1520, Springer-Verlag, Berlin, 1998, pp. 40–54.
4. Banâtre, J.-P. and Le Métayer, D., The GAMMA Model and Its Discipline of Programming. *Science of Computer Programming*, Vol. 15, No. 1 (1990), pp. 55–77.
5. de Boer, F. S., Gabbrielli, M., Marchiori, E. and Palamidessi, C., Proving Concurrent Constraint Programs Correct. In *Conf. Record of the 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, ACM Press, 1994, pp. 98–108.
6. Cardelli, L. and Gordon, A. D., Mobile Ambients. In *Foundations of Software Science and Computational Structures*, Maurice Nivat (ed.), LNCS 1378, Springer-Verlag, Berlin, 1998, pp. 140–155.
7. Chandy, K. M. and Taylor, S., *An Introduction to Parallel Programming*. Jones and Bartlett, Boston, 1992.

8. Chandy, K. M. and Kesselman, C., CC++: A Declarative Concurrent Object-Oriented Programming Notation. In *Research Directions in Concurrent Object-Oriented Programming*, Agha, G., Wegner, P. and Yonezawa, A. (eds.), The MIT Press, Cambridge, MA, 1993, pp. 281–313.
9. Chikayama, T. and Kimura, Y., Multiple Reference Management in Flat GHC. In *Proc. 4th Int. Conf. on Logic Programming (ICLP'87)*, The MIT Press, Cambridge, MA, 1987, pp. 276–293.
10. Chikayama, T., Sato, H. and Miyazaki, T., Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, Tokyo, 1988, pp. 230–251.
11. Chikayama, T., Operating System PIMOS and Kernel Language KL1. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha and IOS Press, Tokyo, 1992, pp. 73–88.
12. Chikayama, T., Fujise, T. and Sekita, D., A Portable and Efficient Implementation of KL1. In *Proc. 6th Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'94)*, LNCS 844, Springer-Verlag, Berlin, 1994, pp. 25–39.
13. Cho, K. and Ueda, K., Diagnosing Non-Well-Moded Concurrent Logic Programs, In *Proc. 1996 Joint Int. Conf. and Symp. on Logic Programming (JICSLP'96)*, The MIT Press, Cambridge, MA, 1996, pp. 215–229.
14. Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.
15. Foster, I. and Taylor, S., Strand: a Practical Parallel Programming Tool. In *Proc. 1989 North American Conf. on Logic Programming (NACLP'89)*, The MIT Press, Cambridge, MA, 1989, pp. 497–512.
16. Foster, I. and Kesselman, C., *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann, San Francisco, 1998.
17. Fournet, C., Gonthier, G. Lévy, J.-J., Maranget, L. and Rémy, D., A Calculus of Mobile Agents. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, LNCS 1119, Springer-Verlag, Berlin, 1996, pp. 406–421.
18. Frühwirth, T., Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.
19. 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 (ICLP'91)*, The MIT Press, Cambridge, MA, 1991, pp. 535–548.
20. Fujita, M., Slaney, J. and Bennett, F., Automatic Generation of Some Results in Finite Algebra. In *Proc. 13th Int. Joint Conf. on Artificial Intelligence (IJCAI'93)*, 1993, pp. 52–57.
21. Haridi, S., Van Roy, P., Brand, P. and Schulte, C., Programming Languages for Distributed Applications. *New Generation Computing*, Vol. 16, No. 3 (1998), pp. 223–261.
22. Hasegawa, R. and Fujita, M., Parallel Theorem Provers and Their Applications. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha and IOS Press, Tokyo, 1992, pp. 132–154.
23. Hirata, K., Yamamoto, R., Imai, A., Kawai, H., Hirano, K., Takagi, T., Taki, K., Nakase, A. and Rokusawa, K., Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha and IOS Press, Tokyo, 1992, pp. 436–459.
24. Hoare, C. A. R., *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

25. Ichiyoshi N., Miyazaki T. and Taki, K., A Distributed Implementation of Flat GHC on the Multi-PSI. In *Proc. 4th Int. Conf. on Logic Programming (ICLP'87)*, The MIT Press, Cambridge, MA, 1987, pp. 257–275.
26. Janson, S. and Haridi, S., Programming Paradigms of the Andorra Kernel Language. In *Proc. 1991 Int. Logic Programming Symp. (ILPS'91)*, The MIT Press, Cambridge, MA, 1991, pp. 167–183.
27. Kahn, K. M., ToonTalk—An Animated Programming Environment for Children. *J. Visual Languages and Computing*, Vol. 7, No. 2 (1996), pp. 197–217.
28. Maher, M. J., Logic Semantics for a Class of Committed-Choice Programs. In *Proc. Fourth Int. Conf. on Logic Programming (ICLP'87)*, The MIT Press, Cambridge, MA, 1987, pp. 858–876.
29. Milner, R., *Communication and Concurrency*. Prentice-Hall International, London, 1989.
30. Milner, R., Parrow, J. and Walker, D., A Calculus of Mobile Processes, I+II. *Information and Computation*, Vol. 100, No. 1 (1992), pp. 1–77.
31. Milner, R., Calculi for Interaction. *Acta Informatica*, Vol. 33, No. 8 (1996), pp. 707–737.
32. Muggleton, S., Inverse Entailment and Progol. *New Generation Computing*, Vol. 13 (1995), pp. 245–286.
33. Naish, L. All Solutions Predicates in Prolog. In *Proc. 1985 Symp. on Logic Programming (SLP'85)*, IEEE, 1985, pp. 73–77.
34. Nakajima K., Inamura Y., Rokusawa K., Ichiyoshi N. and Chikayama, T., Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. Sixth Int. Conf. on Logic Programming (ICLP'89)*, The MIT Press, Cambridge, MA, 1989, pp. 436–451.
35. Nitta, K., Taki, K. and Ichiyoshi, N., Experimental Parallel Inference Software. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha and IOS Press, Tokyo, 1992, pp. 166–190.
36. Petri, C.A., Fundamentals of a Theory of Asynchronous Information Flow. In *Proc. IFIP Congress 62*, North-Holland Pub. Co., Amsterdam, 1962, pp.386–390.
37. Santos Costa V., Warren, D. H. D. and Yang, R., Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Proc. Third ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP'91)*, SIGPLAN Notices, Vol. 26, No. 7 (1991), pp. 83–93.
38. Saraswat, V. A. and Rinard, M., Concurrent Constraint Programming (Extended Abstract). In *Conf. Record of the Seventeenth Annual ACM Symp. on Principles of Programming Languages*, ACM Press, 1990, pp. 232–245.
39. Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conference on Logic Programming (NACLP'90)*, The MIT Press, Cambridge, MA, 1990, pp. 431–446.
40. Saraswat, V. A., Rinard, M. C. and Panangaden, P., Semantic Foundations of Concurrent Constraint Programming. In *Conf. Record of the Eighteenth Annual ACM Symp. on Principles of Programming Languages*, ACM Press, 1991, pp. 333–352.
41. Sato, T. and Tamaki, H., First Order Compiler: A Deterministic Logic Program Synthesis Algorithm. *J. Symbolic Computation*, Vol. 8, No. 6 (1989), pp. 605–627.
42. Shapiro, E. Y., Concurrent Prolog: A Progress Report. *IEEE Computer*, Vol. 19, No. 8 (1986), pp. 44–58.
43. Shapiro, E., The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.

44. 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.
45. Schulte, C. and Smolka, G., Encapsulated Search for Higher-order Concurrent Constraint Programming. In *Proc. 1994 International Logic Programming Symp. (ILPS'94)*, The MIT Press, Cambridge, MA, 1994, pp. 505–520.
46. Smolka, G., The Oz Programming Model. In *Computer Science Today*, van Leeuwen, J. (ed.), LNCS 1000, Springer-Verlag, Berlin, 1995, pp. 324–343.
47. Somogyi, Z., Henderson, F. and Conway, T., The Execution Algorithm of Mercury, An Efficient Purely Declarative Logic Programming Language. *J. Logic Programming*, Vol. 29, No. 1–3 (1996), pp. 17–64.
48. Taki, K., Parallel Inference Machine PIM. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, Ohmsha and IOS Press, Tokyo, 1992, pp. 50–72.
49. Tick, E. The Deevolution of Concurrent Logic Programming Languages. *J. Logic Programming*, Vol. 23, No. 2 (1995), pp. 89–123.
50. Toyoda, M., Shizuki, B., Takahashi, S., Matsuoka, S. and Shibayama, E., Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment. In *Proc. IEEE Symp. on Visual Languages*, IEEE, 1997, pp. 76–83.
51. Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming '85*, Wada, E. (ed.), LNCS 221, Springer-Verlag, Berlin, 1986, pp. 168–179.
52. Ueda, K. and Chikayama, T., Concurrent Prolog Compiler on Top of Prolog. In *Proc. 1985 Symp. on Logic Programming (SLP'85)*, IEEE, 1985, pp. 119–126.
53. Ueda, K. Making Exhaustive Search Programs Deterministic. In *Proc. Third Int. Conf. on Logic Programming (ICLP'86)*, LNCS 225, Springer-Verlag, Berlin, 1986, pp. 270–282. Revised version in *New Generation Computing*, Vol. 5, No. 1 (1987), pp. 29–44.
54. 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.
55. Ueda, K., Parallelism in Logic Programming. In *Information Processing 89, Proc. IFIP 11th World Computer Congress*, North-Holland/IFIP, 1989, pp. 957–964.
56. 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.
57. Ueda, K. and Chikayama, T. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.
58. Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming (ICLP'90)*, The MIT Press, Cambridge, MA, 1990, pp. 3–17. Revised version in *New Generation Computing* [61].
59. Ueda, K., Designing a Concurrent Programming Language. In *Proc. InfoJapan'90*, Information Processing Society of Japan, Tokyo, 1990, pp. 87–94.
60. Ueda, K. and Morita, M., Message-Oriented Parallel Implementation of Moded Flat GHC. *New Generation Computing*, Vol. 11, No. 3–4 (1993), pp. 323–341.
61. Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.
62. Ueda, K., Moded Flat GHC for Data-Parallel Programming. In *Proc. FGCS'94 Workshop on Parallel Logic Programming*, ICOT, Tokyo, 1994, pp. 27–35.

63. Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, LNCS 1068, Springer-Verlag, Berlin, 1996, pp. 134–153.
64. Ueda, K., Linearity Analysis of Concurrent Logic Programs. In preparation.