Concurrent Logic Programming: Met and Unmet Promises

Iliano Cervesato[1] and Edmund S.L. Lam[2]

[1] Carnegie Mellon University
iliano@cmu.edu
[2] University of Colorado, Boulder
edmund.lam@colorado.edu

Abstract

Logic programming has been heralded as the quintessential declarative programming paradigm, although many instances provide extra-logical constructs that undermine this aspiration. The word "declarative" conjures two promises: the first is the ability to write code that reflects a natural, human-friendly, description of the problem at hands as opposed to a mechanistic, hardware-oriented, encoding of a solution. The second is the opportunity to reason logically about it, thereby automatically strengthening assurance, security and performance. While the first promise has been fairly successful in some domains, the second still has to live to its expectations. We explore both promises in the context of concurrent logic programming. We highlight them using CoMingle, a logic programming language designed to develop mobile Android applications.

1 Logical Specification of Concurrent Applications

Concurrent and distributed applications have traditionally been developed by writing a separate piece of code for each participating device (or class of devices). This *node-centric* approach puts the onus of handling messaging and synchronization on the programmer. This is no small burden: on the messaging side, the programmer needs to make sure that each sent message has a recipient and vice versa, and that sender and receiver agree on its format — simple tasks that quickly become a time sink as an application grows larger. The synchronization side is more tricky as the programmer is left alone battling the many pitfalls of concurrency (deadlocks, live locks and unwanted race conditions) — complex tasks even for small applications. Because the code running on each device is a separate control flow, little automation is available to alleviate these concerns as current program analysis techniques typically focus on individual control-flows and do not work well for reasoning about the executions of a concurrent program as a whole. These effects are compounded by the fact that, as mobile applications become commonplace, many of them are being developed by programmers with relatively little training or experience.

An alternative approach is to write a unified program that captures the behavior of a distributed application as a single entity. This *system-centric* approach gives the programmer a bird-eye's view of the behavior he/she is trying to achieve. Being a single program, it is easier to automate basic checks such as message format consistency, for example as a form of type-checking. This unitary system-centric specification is automatically transformed into the node-centric code that runs on actual devices through a process called choreographic compilation. It is this transformation, rather than the programmer, that handles the tedium of managing communication and the intricacies of getting synchronization right.

While the system-centric approach to programming distributed application is not exclusive to the paradigm of logic programming (in fact, it underlies many of Google's applications [4]), logic programming is proving particularly well-suited for this purpose [1, 5, 9, 10]. Logic programming provides a natural way to write specifications that represent how the distributed computation proceeds as a whole rather than forcing the point of view of any

specific node. One language that embraces this philosophy is CoMingle [9]. CoMingle is a rule-based language for programming mobile distributed applications, originally Android apps. CoMingle implements a fragment of first-order linear logic using a forward-chaining semantics, as found in languages based on multiset-rewriting such as CHR [3]. It enriches it with sorts (making it a strongly-typed polymorphic language), locations (which identify computing nodes), and multiset comprehensions (which provide a natural mechanism to manipulate arbitrarily many facts matching a given pattern). Specifically designated atomic facts allow CoMingle to trigger local computations and respond to them (used for example to process input from an Android device or to render output on the screen). We used an advanced prototype of CoMingle [7] to implement a number of mobile applications. We were able to write each of them in a few hours, which compares favorably with the standard node-centric approach. We built one such application both using CoMingle and by writing traditional code: the former was about one tenth of the size of the latter with no noticeable difference in performance [8]. This ease of development gave us time to experiment with application-level features, with new communication behaviors typically taking minutes to implement.

2 Reasoning about Concurrent Applications

Because in its purest form a program is a logical formula, logic programming has often been trumpeted as facilitating reasoning about one's code, where reasoning is variedly understood as providing provable assurances of correctness, guaranteed performance, and more recently security. With a few exceptions (e.g., [11] about performance bounds), we argue that such expectations of correctness have not been met. For example, correctness presupposes a specification that can be compared with an implementation, but rarely does a programmer write two such formulas for the same problem, and in any case tools to verify the expected subsumption are rarely available.

Concurrent logic programs, for example the ones we wrote in CoMingle, similarly come short of availing themselves of the reasoning possibilities of the underlying logic. The consequences are somewhat more dire in this setting as writing concurrent programs is much harder than developing code that does not engage in synchronization. The proliferation and ease of deployment of mobile apps, again often developed by novices, means that there is a lot of buggy code out there, with much more to come.

Even in a large program, a fairly small part of the code of a distributed application is about concurrent interactions, often with recurring patterns (this is particularly evident in CoMingle programs, where inter-node communication and local computation are written in separate languages — CoMingle itself and Java, respectively). We postulate that this is an opportunity for logic-based methods, if not wholesale logical reasoning, to participate in the development of concurrent and distributed applications in the form of formal analysis tools. One promising idea is session types [6], which describe the communication pattern of a program, thereby allowing the implementation of a tool to statically catch messaging errors and deadlocks. Session types are currently limited to relatively simple interactions, but they are rapidly being developed to handle larger classes. Other techniques include logic-based modularity [2], which gives the programmer control over the scope of interactions (in contrast to the traditionally flat name-space of logic programming). One last class of techniques that holds substantial promises in the development of correct concurrent programs specifically is coinductive reasoning, for example in the form of bisimulation. While tools are still in their infancy, the growing realization that many program properties are coinductive in nature are sure to accelerate their development.

What logic programming does is to give the programmer a language that abstracts some idiosyncrasies of the underlying machine, ultimately letting him/her write less code: abstraction, not reasoning, makes small programs easier to get right. But as programs grow,

coding complexity creeps back up, with little available to the programmer to manage it in a typical logic programming language. We postulate that the largely untapped reasoning potential of logic programming in general, and concurrent logic programming in particular, holds the promise provide assurances that is elusive in other paradigms.

References

[1] M. P. Ashley-Rollman et al. A Language for Large Ensembles of Independently Executing Nodes. In *ICLP'09*, pages 265–280. Springer LNCS 5649, 2009.

[2] I. Cervesato and E. S. Lam. Modular Multiset Rewriting. In *Proceedings of LPAR'15*, pages 1–17, Suva, Fiji, mber. Springer LNCS 9450.

[3] T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[4] Google Inc. Google Web Toolkit. Available at http://code.google.com/webtoolkit/.

[5] S. Grumbach and F. Wang. Netlog, a Rule-based Language for Distributed Programming. In *PADL'10*, pages 88–103. Springer LNCS 5937, 2010.

[6] K. Honda. Types for dyadic interaction. In *Proceedings of CONCUR'93*, pages 509–523. Springer LNCS 715, 1993.

[7] E. S. Lam. CoMingle: Distributed Logic Programming Language for Android Mobile Ensembles. Download at `https://github.com/sllam/comingle`, 2014.

[8] E. S. Lam and I. Cervesato. Comingle: Distributed Logic Programming for Decentralized Android Applications. Technical Report CMU-CS-15-101, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2015.

[9] E. S. Lam, I. Cervesato, and N. F. Haque. Comingle: Distributed Logic Programming for Decentralized Mobile Ensembles. In *COORDINATION'15*. Springer LNCS 9037, 2015.

[10] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD'06*, pages 97–108, 2006.

[11] D. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, July 2002.