Convergence in Language Design: A Case of Lightning Striking Four Times in the Same Place

Peter Van Roy

Université catholique de Louvain,
B-1348 Louvain-la-Neuve, Belgium
pvr@info.ucl.ac.be
http://www.info.ucl.ac.be/people/cvvanroy.html

Abstract. What will a definitive programming language look like? By *definitive language* I mean a programming language that gives good solutions at its level of abstraction, allowing computer science researchers to move on and work at higher levels. Given the evolution of computer science as a field with a rising level of abstraction, it is my belief that a small set of definitive languages will eventually exist. But how can we learn something about this set, considering that many basic questions about languages have not yet been settled? In this paper, I give some tentative conclusions about one definitive language. I present four case studies of substantial research projects that tackle important problems in four quite different areas: fault-tolerant programming, secure distributed programming, network-transparent distributed programming, and teaching programming as a unified discipline. All four projects had to think about language design. In this paper, I summarize the reasons why each project designed the language it did. It turns out that all four languages have a common structure. They can be seen as layered, with the following four layers in this order: a strict functional core, then deterministic concurrency, then message-passing concurrency, and finally shared-state concurrency (usually with transactions). This confirms the importance of functional programming and message passing as important defaults; however, global mutable state is also seen as an essential ingredient.

1 Introduction

This paper presents a surprising example of convergence in language design.[1] I will present four different research projects that were undertaken to solve four very different problems. The solutions achieved by all four projects are significant contributions to each of their respective areas. The four projects are interesting to us because they all considered language design as a key factor to achieve success. The surprise is that the four projects ended up using languages that have very similar structures.

[1] This paper was written to accompany an invited talk at FLOPS 2006 and is intended to stimulate discussion.

M. Hagiya and P. Wadler (Eds.): FLOPS 2006, LNCS 3945, pp. 2–12, 2006.

This paper is structured as follows. Section 1.1 briefly presents each of the four projects and Section 1.2 sketches their common solution. Then Sections 2 to 5 present each of the four projects in more detail to motivate why the common solution is a good solution for it. Finally, Section 6 concludes the paper by recapitulating the common solution and making some conclusions on why it is important for functional and logic programming.

Given the similar structure of the four languages, I consider that their common structure deserves to be carefully examined. The common structure may turn out to be the heart of one possible *definitive* programming language, i.e., a programming language that gives good solutions at its level of abstraction, so that computer science researchers can move on and work at higher levels. My view is that the evolution of programming languages will follow a similar course as the evolution of parsing algorithms. In the 1970s, compiler courses were often built around a study of parsing algorithms. Today, parsing is well understood for most practical purposes and when designing a new compiler it is straightforward to pick a parsing algorithm from a set of "good enough" or "definitive" algorithms. Today's compiler courses are built around higher level topics such as dataflow analysis, type systems, and language design. For programming languages the evolution toward a definitive set may be slower than for parsing algorithms because languages are harder to judge objectively than algorithms.

1.1 The Four Projects

The four projects are the following:[2]

- Programming highly available embedded systems for telecommunications (Section 2). This project was undertaken by Joe Armstrong and his colleagues at the Ericsson Computer Science Laboratory. This work started in 1986. The Erlang language was designed and a first efficient and stable implementation was completed in 1991. Erlang and its current environment, the OTP (Open Telecom Platform) system, are being used successfully in commercial systems by Ericsson and other companies.
- Programming secure distributed systems with multiple users and multiple security domains (Section 3). This project was undertaken over many years by different institutions. It started with Carl Hewitt's Actor model and led via concurrent logic programming to the E language designed by Doug Barnes, Mark Miller, and their colleagues. Predecessors of E have been used to implement various multiuser virtual environments.
- Making network-transparent distributed programming practical (Section 4). This project started in 1995 with the realization that the well-factored design of the Oz language, first developed by Gert Smolka and his students in 1991 as an outgrowth of the ACCLAIM project, was a good starting point for making network transparent distribution practical. This resulted in the Mozart Programming System, whose first release was in 1999.

[2] Many people were involved in each project; because of space limitations only a few are mentioned here.

– Teaching programming as a unified discipline covering all popular programming paradigms (Section 5). This project started in 1999 with the realization by the author and Seif Haridi that Oz is well-suited to teaching programming because it covers many programming concepts, it has a simple semantics, and it has an efficient implementation. A textbook published in 2004 "reconstructs" the Oz design according to a principled approach. This book is the basis of programming courses now being taught at more than a dozen universities worldwide.

1.2 The Layered Language Structure

In all four research projects, the programming language has a layered structure. In its most general form, the language has four layers. This section briefly presents the four layers and mentions how they are realized in the four projects. The rest of the paper motivates the layered structure for each project in more detail. The layers are the following:

– The inner layer is a strict functional language. All four projects start with this layer.
– The second layer adds deterministic concurrency. Deterministic concurrency is sometimes called declarative or dataflow concurrency. It has the property that it cannot have race conditions. This form of concurrency is as simple to reason in as functional programming. In Oz it is realized with single-assignment variables and dataflow synchronization. Because Oz implements these variables as logic variables, this layer in Oz is also a logic language. In E it is realized by a form of concurrent programming called *event-loop concurrency*: inside a process all objects share a single thread. This means that execution inside a process is deterministic. The Erlang project skips this layer.
– The third layer adds asynchronous message passing. This leads to a simple message-passing model in which concurrent entities send messages asynchronously. All four projects have this layer. In E, this layer is used for communication between processes (deterministic concurrency is used for communication inside a single process).
– The fourth layer adds global mutable state.[3] Three of the four projects have global mutable state as a final layer, provided for different reasons, but always with the understanding that it is not used as often as the other layers. In the Erlang project, the mutable state is provided as a persistent database with a transactional interface. In the network transparency project, the mutable state is provided as an object store with a transactional interface and as a family of distributed protocols that is used to guarantee coherence of state across the distributed system. These protocols are expensive but they are sometimes necessary. In the teaching programming project, mutable state is used to make programs modular. The E project skips this layer.

[3] By *global*, I mean that the mutable state has a scope that is as large as necessary, not that it necessarily covers the whole program.

This layered structure has an influence on program design. In all four projects, the starting point is the functional inner layer, complemented by the message-passing layer which is just as important. In three of the four projects, the final layer (global mutable state) is less used than the others, but it provides a critical functionality that cannot be eliminated.

Note that the network-transparent distribution project and the teaching programming project were undertaken by many of the same people and started with the same programming language. Both projects were undertaken because we had reasons to believe Oz would be an adequate starting point. Each project had to adapt the Oz language to get a good solution. In the final analysis, both projects give good reasons why their solutions are appropriate, as explained in Sections 4 and 5.

2 Fault-Tolerant Programming

The Erlang programming language and system is designed for building high availability telecommunications systems. Erlang was designed at the Ericsson Computer Science Laboratory [5, 4]. Erlang is designed explicitly to support programs that tolerate both software and hardware faults. Note that software faults are unavoidable: studies have shown that even with extensive testing, software still has bugs. Any system with high availability must therefore have a way to tolerate faults due to software bugs. Erlang has been used to build commercial systems of very high availability [8]. The most successful of these systems is the AXD 301 ATM switch, which contains around 1 million lines of Erlang, a similar amount of C/C++ code, and a small amount of Java [29].

An Erlang program consists of a (possibly very large) number of processes. An Erlang process is a lightweight entity with its own memory space. A process is programmed with a strict functional language. Each process has a unique identity, which is a constant that can be stored in data structures and in messages. Processes communicate by sending asynchronous messages to other processes. A process receives messages in its mailbox, and it can extract messages from the mailbox with pattern matching. Note that a process can do dynamic code change by receiving a new function in a message and installing it as the new process definition. We conclude that this structure gives the Erlang language two layers: a functional layer for programming processes, and a message-passing layer for allowing them to communicate.

To support fault tolerance, two processes can be linked together. When one process fails, for example because of a software error, then the other fails as well. Each process has a supervisor bit. If a process is set to supervisor mode, then it does not fail when a linked process fails, but it receives a message generated by the run-time system. This allows the application to recover from the failure. Erlang is well-suited to implement software fault tolerance because of process isolation and process linking.

Erlang also has a database called Mnesia. The database stores consistent snapshots of critical program data. When processes fail, their supervisors can use the database to recover and continue execution. The database provides a

transactional interface to shared data. The database is an essential part of Erlang programs. It can therefore be considered as a third layer of the Erlang language. This third layer, mutable state with a transactional interface, implements a form of shared-state concurrency [26].

Because Erlang processes do not share data, they can be implemented over a distributed system without any changes in the program. This makes distributed programming in Erlang straightforward. Using process linking and supervisors, Erlang programs can also recover from hardware failures, i.e., partial failures of the distributed system.

3 Secure Distributed Programming

The E programming language and system is designed for building secure distributed systems [21, 19]. The E language consists of objects (functions that share an encapsulated state) hosted in secure processes called *vats* that communicate through a secure message-passing protocol based on encryption. Within the language, security is provided by implementing all language references (including object references) as capabilities. A *capability* is an unforgeable reference that combines two properties that cannot be separated: it designates a language entity and it provides permission to perform a well-defined set of operations on the entity. The only way to perform an operation is to have a capability for that operation.

Capabilities are passed between language entities according to well-defined rules. The primary rule is that the only way to get a capability is that an entity to which you already have a capability passes you the capability ("connectivity begets connectivity"). A system based on capabilities can support the Principle of Least Authority (POLA): give each entity just enough authority to carry out its work. In systems based on POLA the destructive abilities of malicious programs such as viruses largely go away. Unfortunately, current programming languages and operating systems only have weak support for POLA. This is why projects such as E and KeyKOS (see below) are so important [25].

Inside a vat, there is a single thread of execution and all objects take turns executing in this thread. Objects send other objects asynchronous messages that are queued for execution. Objects execute a method when they receive a message. This is a form of deterministic concurrency that is called *event-loop concurrency*. Single threading within a vat is done to ensure that concurrency introduces no security problems due to the nondeterminism of interleaving execution. Event-loop concurrency works well for secure programs; a model based on shared-state concurrency is much harder to program with [20]. Between two or more vats, execution is done according to a general message-passing model.

In a system such as E that is based on capabilities, there is no *ambient authority*, i.e., a program does not have the ability to perform an operation just because it is executing in a certain context. This is very different from most other systems. For example, in Unix a program has all the authority of the user that executes it. The lack of ambient authority does not mean that E necessarily does not have global mutable state. For example, there could be a capability

that is given by default to all new objects. However, the current design of E does not have global mutable state. If information needs to be shared globally, the sharing is programmed explicitly by using message-passing concurrency.

The history of E starts with Carl Hewitt's Actor model in the 1970s [13, 14] and continues with Norm Hardy's KeyKOS system [10], which is a pure capability operating system that provides orthogonal persistence. It continues with the Concurrent Prolog family of languages [23]. The Joule language, designed at Agorics [1], is E's most direct ancestor. E was originally designed at Electric Communities as an infrastructure for building a secure distributed collaborative computing environment, secure enough that you could spend real money and sign real contracts within it. Virtual environments now exist with currencies that are exchangeable with real currencies; they are called *virtual economies* [30].

4 Network-Transparent Distributed Programming

This project was motivated by the desire to simplify distributed programming by making a practical system that is both network transparent and network aware. This approach was first expressed clearly by Cardelli in his work on Obliq [6]. The idea is to make a distributed implementation of a language by implementing the basic language operations with distributed algorithms. By choosing the algorithms carefully, the implementation can be made efficient and can handle partial failure inside the language [12]. A program then consists of two separate parts: the functionality, in which distribution is ignored, and the choice of distributed algorithms, which is used to tune network performance and to handle partial failure. We are extending this approach to handle security [24].

Some researchers have maintained that this cannot work; that network transparency cannot be made practical, see, e.g., Waldo *et al* [28]. They cite four reasons: pointer arithmetic, partial failure, latency, and concurrency. The first reason (pointer arithmetic) disappears if the language has an abstract store. The second reason (partial failure) requires a reflective fault model, which we designed for the Distributed Oz language. The final two reasons (latency and concurrency) lead to a layered language design. Let us examine each of these reasons. Latency is a problem if the language relies primarily on synchronized operations. In the terminology of Cardelli, latency is a network awareness issue. The solution is that the language must make asynchronous programming both simple and efficient.

Concurrency is a problem if the language relies heavily on mutable state. To achieve network transparency, the mutable state has to be made coherent across all the machines of the system. It is well known that this is costly to achieve for a distributed system. The solution is to avoid the use of mutable state as much as possible, and to use it only when it is absolutely necessary. As a result, most of the program is concurrent and functional. Global state is necessary only in a few places, e.g., to implement servers and caches, and in general it can be avoided (note that local state, which is limited to a single machine, is fine).

Our distributed programming language therefore has a layered structure. The core has no state and is therefore a functional language. Extending the func-

tional language with concurrency and a simple communications channel gives multi-agent programming or actor programming: concurrent entities that send each other asynchronous messages. The final step is to add mutable state, with a choice of protocols for its implementation. For example, stationary state corresponds to a standard server architecture. Mobile or cached state can be used to increase performance by moving the state to where it is currently being used [27]. Other protocols are possible too. We find that a good way to add mutable state is as part of a transactional protocol [3]. Transactions are a good way to hide both network latency and partial failure.

The final language is organized into four layers, in this order: a strict functional core, dataflow concurrency, communication channels, and mutable state. For language entities in each layer, distributed algorithms implement the distributed behavior. Inner layers have more efficient distributed behaviors. We implement dataflow concurrency with single-assignment variables, which are intermediate between no assignment (functional language) and any number of assignments (mutable state). Single-assignment variables are implemented with a distributed unification algorithm, which is more efficient than a state coherence protocol [11]. To write an efficient distributed program, one uses the lower layers preferentially and one chooses the appropriate distributed algorithm for each language entity that is distributed. Partial failure is handled at the language level by asynchronous notifications similar to the process linking provided by Erlang.

5 Teaching Programming as a Unified Discipline

A good way to organize a programming course is to start with a simple language and then to extend this language gradually. This organization was pioneered in the 1970s by Holt *et al*, who used carefully defined subsets of PL/I [15]. The most successful application of this organization was done by Abelson & Sussman in 1985, who use subsets of Scheme and start with a simple functional language [2]. A simple functional language is a good start for teaching programming, for many reasons. It is easy to explain because of its simple semantics and syntax, and yet it contains a key language concept, the lexically scoped closure, which is the basis for many other powerful concepts.

Abelson & Sussman made the important decision to organize the subsets according to the programming concepts they contain, and not the language features they use as Holt did. This makes the course less dependent on the details of one language and gives students a broader and more in-depth understanding. The second concept introduced by Abelson & Sussman is mutable state. With mutable state it is possible to express the object-oriented programming style, with an object as a collection of functions accessing a common mutable state that is hidden through lexical scoping. Unfortunately, by introducing mutable state early on, programs in the new language are no longer mathematical functions. This makes reasoning about programs harder.

In 1999, the author and Seif Haridi realized that they understood programming concepts well enough to teach programming in a more unified way than had

been done before. We chose the Oz language because of its well-factored design. We set about writing a programming textbook and organizing courses [26]. During this work, we "reconstructed" the design of a large subset of Oz according to an organizing principle that states that a new concept is needed when programs start getting complicated for reasons unrelated to the problem being solved. More precisely, a new concept is needed in the language when programs require nonlocal transformations to encode the concept in the language. If the new concept is added to the language, then only local transformations are needed. We call this the *creative extension principle*. It was first defined by Felleisen [9].

We found that it is possible to add concurrency as the second concept instead of mutable state. The resulting language lets us write purely functional programs as collections of independent entities ("agents") that communicate through deterministic streams. This form of concurrency is called *declarative concurrency*. The streams are deterministic because the writer and readers of each stream element are known deterministically. The difference with a sequential functional language is that the output of a function can be calculated incrementally instead of all at once. Race conditions are not possible, i.e., there is no observable nondeterminism in the language.

Declarative concurrency is a deterministic form of concurrency that is much simpler to program with than the shared-state concurrency used in mainstream languages such as Java [18]. It is already widely used, e.g., Unix pipes and Google's MapReduce [7] are just two of many examples, but it is not well-known as a programming model. Because of its simplicity we consider that it deserves to become more popular. For example, Morrison shows how to use it for business software [22]. We have taught declarative concurrency as a first introduction to concurrent programming in second-year university courses at several large universities.

After introducing concurrency, the next concept we introduce is a simple communication channel. This extends the previous model by adding nondeterminism: the writer of the next stream element is chosen nondeterministically among the potential writers. The resulting language is both practical and easy to program in [16].

Finally, we introduce global mutable state. This is important for program modularity, i.e., the ability to change part of a program without having to change the rest. Without true mutable state, modularity is not possible [26]. State-threading techniques such as monads are not expressive enough [17].

6 Conclusions

This paper presents four successful research projects that were undertaken to solve quite different problems, namely fault-tolerant programming, secure distributed programming, network-transparent distributed programming, and teaching programming as a unified discipline. Each project had to consider language design to solve its problem. A surprising result is that the four resulting languages have a common structure. In the general case they are layered, with a strict functional inner layer, a deterministic concurrency layer, a message-

passing concurrency layer, and a shared-state concurrency layer, in that order. I postulate that this common structure will be part of one possible definitive programming language, i.e., a programming language that gives good enough solutions at its level of abstraction so that computer scientists and developers can move on to higher levels.

Given this postulate one can deduce several important consequences for functional and logic programming. First, that the notion of declarative programming, i.e., functional and logic programming, is at the very core of programming languages. This is already well-known; our study reinforces this conclusion. Second, that declarative programming will stay at the core for the foreseeable future, because distributed, secure, and fault-tolerant programming are essential topics that need support from the programming language. A third conclusion is that it is important for declarative programmers to study how declarative programming fits in the larger scheme. A final conclusion is that message-passing concurrency seems to be the correct default for general-purpose concurrent programming instead of shared-state concurrency.

Acknowledgments

We would like to thank Kevin Glynn and Boris Mejias for their comments on a draft of this article. We would like to thank the members of the Programming Languages and Distributed Computing group at UCL for creating the environment in which the speculations of this article could arise. This work was partly funded by the EVERGROW project in the sixth Framework Programme of the European Union under contract number 001935 and by the MILOS project of the Wallonia Region of Belgium under convention 114856.

References

1. Agorics, Inc., 2004. `www.agorics.com`.
2. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985. Second edition 1996.
3. Mostafa Al-Metwally. *Design and Implementation of a Fault-Tolerant Transactional Object Store*. PhD thesis, Al-Azhar University, Cairo, Egypt, December 2003.
4. Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology (KTH), Kista, Sweden, November 2003.
5. Joe Armstrong, Mike Williams, Claes Wikström, and Robert Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1996.
6. Luca Cardelli. A language with distributed scope. In *Principles of Programming Languages (POPL)*, pages 286–297, San Francisco, CA, January 1995. ACM Press.
7. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 137–150, December 2004.

8. Ericsson. *Open Telecom Platform—User's Guide, Reference Manual, Installation Guide, OS Specific Parts*. Telefonaktiebolaget LM Ericsson, Stockholm, Sweden, 1996.
9. Matthias Felleisen. On the expressive power of programming languages. In *3rd European Symposium on Programming (ESOP 1990)*, pages 134–151, May 1990.
10. Norman Hardy. KeyKOS architecture. In *ACM SIGOPS Operating Systems Review*, volume 19, pages 8–25, October 1985.
11. Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.
12. Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, May 1998.
13. Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.
14. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245, August 1973.
15. R.C. Holt, D.B. Wortman, D.T. Barnard, and J.R. Cordy. SP/k: A system for teaching computer programming. *Communications of the ACM*, 20(5):301–309, May 1977.
16. Sverker Janson, Johan Montelius, and Seif Haridi. Ports for Objects in Concurrent Logic Programs. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Trends in Object-Based Concurrent Computing*, pages 211–231. MIT Press, Cambridge, MA, 1993.
17. Lambda the Ultimate discussion. State and modularity, October 2003. Available at `lambda-the-ultimate.org/classic/message9361.html`.
18. Doug Lea. *Concurrent Programming in Java,* 2nd edition. Addison-Wesley, 2000.
19. Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Proceedings of the 4th International Conference on Financial Cryptography*, volume 1962 of *Lecture Notes in Computer Science*, pages 349–378. Springer-Verlag, 2000.
20. Mark S. Miller and Jonathan Shapiro. Concurrency among strangers. In *Proceedings of the Symposium on Trustworthy Global Computing (TGC 2005)*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. Springer-Verlag, April 2005.
21. Mark S. Miller, Marc Stiegler, Tyler Close, Bill Frantz, Ka-Ping Yee, Chip Morningstar, Jonathan Shapiro, Norm Hardy, E. Dean Tribble, Doug Barnes, Dan Bornstien, Bryce Wilcox-O'Hearn, Terry Stanley, Kevin Reid, and Darius Bacon. E: Open source distributed capabilities, 2001. Available at `www.erights.org`.
22. J. Paul Morrison. *Flow-Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, New York, 1994.
23. Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.
24. Fred Spiessens and Peter Van Roy. The Oz-E project: Design guidelines for a secure multiparadigm programming language. In *Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004*, volume 3389 of *Lecture Notes in Computer Science*, pages 21–40. Springer-Verlag, 2005.
25. Marc Stiegler. The SkyNet virus: Why it is unstoppable; How to stop it. Talk available at `www.erights.org/talks/skynet/`.

26. Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, Cambridge, MA, 2004.
27. Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.
28. Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *Second International Workshop on Mobile Object Systems–Towards the Programmable Internet*, pages 49–64, July 1996. Originally published at Sun Microsystems Laboratories in 1994.
29. Ulf Wiger. Four-fold increase in productivity and quality – industrial-strengh functional programming in telecom-class products. In *Proceedings of the 2001 Workshop on Formal Design of Safety Critical Embedded Systems*, 2001.
30. Wikipedia, the free encyclopedia. Entry "virtual economy", January 2006. Available at `en.wikipedia.org/wiki/Virtual_economy`.