Teaching Programming Broadly and Deeply: The Kernel Language Approach

Peter Van Roy
Department of Computing Science and Engineering (INGI)
Université catholique de Louvain (UCL)
B-1348 Louvain-la-Neuve Belgium
pvr@info.ucl.ac.be

Seif Haridi
Department of Microelectronics and Information Technology (IMIT)
Royal Institute of Technology (KTH)
S-164 28 Kista Sweden
seif@it.kth.se

Abstract: We present the *kernel language approach;* a new way to teach programming that situates most of the widely known programming paradigms (including imperative, object-oriented, concurrent, logic, and functional) into a uniform setting that shows their deep relationships and how to use them together. Widely different practical languages (exemplified by Java, Haskell, Prolog, and Erlang) with their rich panoplies of abstractions and syntax are explained by straightforward translations into closely related *kernel languages*, simple languages that consist of small numbers of *programmer-significant* concepts. Kernel languages are easy to understand and have a simple formal semantics that can be used by practicing programmers to reason about correctness and complexity.

Key words: computer programming, education, curriculum, science, engineering, programming concepts, programming paradigms, kernel language, dataflow, concurrency, functional programming, object-oriented programming, logic programming, imperative programming, programming techniques

---

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35619-8_15
L. Cassel et al. (eds.), *Informatics Curricula and Teaching Methods*

1. EXISTING APPROACHES

For the purposes of this paper, let us consider a broad definition of computer programming as bridging the gap between the specification and the running program. This consists of designing the architecture and abstractions of an application and coding them in a programming language. Programming as a discipline has two essential parts: a technology and its scientific foundation. The technology consists of tools, practical techniques, and standards, allowing one to *do* programming. The science consists of a broad and deep theory with predictive power, allowing one to *understand* programming. The science should be designed to be useful to the practical programmer, to allow him or her to reason about correctness and complexity of practical programs.

Surprisingly, we find that programming is not taught in this way. Rather, it is taught in two different ways: either as a craft in the context of a single programming paradigm and its tools, or as a branch of mathematics. The science is either limited to the chosen paradigm or too fundamental to be of practical use. Let us survey the main focus of existing textbooks: object-oriented programming [15, 21, 24, 25,] imperative programming [22], functional programming [6, 10, 13, 18, 23], logic programming [7, 31], functional/imperative programming [1, 14, 16], and concurrent imperative programming [3, 4]. Some textbooks are explicitly limited to a particular language [4, 5, 7, 9, 20, 21 24, 25, 31, 32]. Almost no attempt is made to put the paradigms in a uniform framework. The only achievement is to show how object-oriented programming can be explained in terms of functional programming and state. A more explicit attempt is Leda [8], but it presents only a few paradigms and does not explain in any depth how to use them together or how they are related. None of these textbooks give a formal semantics except for those on functional programming and concurrent imperative programming. Specialized books on language semantics are either too formal or too restricted for the wider concerns of practical programming [12, 19, 27, 34].

Taken together, these textbooks show programming as a discipline lacking unity. This has a detrimental effect on programmer competence and thus on program quality. A concrete example illustrating this is concurrent programming in Java. Concurrency with shared state and monitors, as used in many languages including Java, is so complicated that it is taught only in advanced courses [20]. Furthermore, the implementation of concurrency in current versions of Java is expensive. Java-taught programmers reach the conclusion that concurrency is *always* complicated and expensive. They write programs to avoid concurrency, often in a contorted way. But these limitations are not fundamental at all; there are useful forms of concurrency,

such as dataflow concurrency (e.g., streams in Unix) and active objects, which are almost as easy to use as sequential programming. Furthermore, it is possible to implement threads almost as cheaply as procedure calls. Teaching concurrency in a broader way would allow programmers to design and program with concurrency in systems without these limitations, including improved implementations of Java.

2. THE KERNEL LANGUAGE APPROACH

How can we teach students the broad view? There are simply too many programming languages to teach them all. Teaching a few carefully selected languages, say one per paradigm (for example Java, Prolog, and Haskell), is a stopgap solution. It multiplies the intellectual effort of the student (since each language has its own syntax and semantics) and does not show the deep connections between the paradigms. The kernel language approach we propose solves these problems.

The approach is not based on a single language (or a few languages), but on the underlying concepts. The concepts are carefully chosen to be meaningful for practical programming. They are organized into simple languages called *kernel languages*. Practical languages in all their richness are translated into the kernel languages in a straightforward manner. This approach is truly language-independent; a wide variety of languages and programming paradigms can be modelled by a small set of closely related kernel languages.

The kernel languages are easy to understand and have a simple formal semantics that allows practicing programmers to reason about correctness and complexity at a high level of abstraction. Programming paradigms and their requisite languages are then emergent phenomena, depending on which concepts one uses. The advantages and limitations of each paradigm show up clearly. This gives the student a deep and broad insight into programming concepts and how to use them to design abstractions. For example, many of our students who were already proficient Java programmers have told us that they first understood what Java objects really were after following our course.

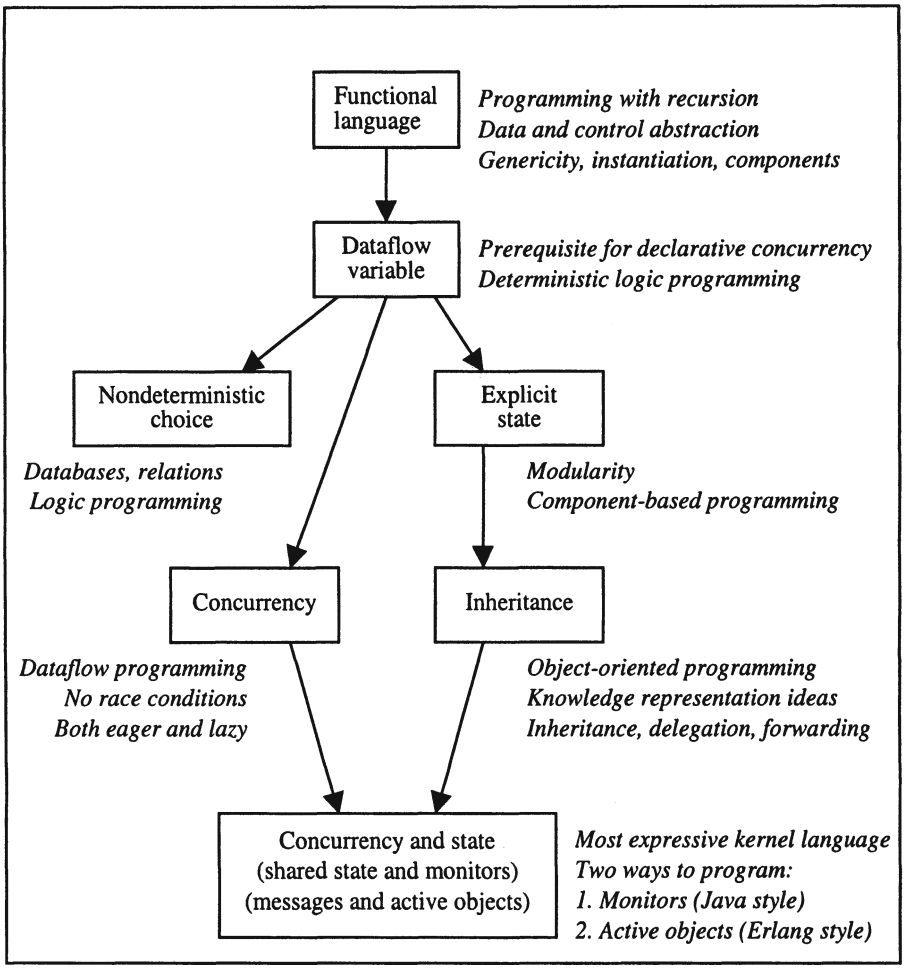

Figure 1. Some important steps in the kernel language approach.

Figure 1 briefly summarizes how we organize programming according to this approach. Each box corresponds to a kernel language. The figure is incomplete; our textbook distinguishes more than 20 paradigms, each with its kernel language. Here are some important steps:

- The most basic kernel language does strict functional programming. This can already express most of the programming techniques of the later kernel languages. It can express data and control abstraction, genericity, instantiation, and components.
- The second kernel language adds dataflow variables (a kind of single-assignment variable). This is essential for two reasons. It is a prerequisite for declarative concurrency. It also means that the second language is a deterministic logic programming language.

- We add nondeterministic choice to the second language. This gives relational programming and nondeterministic logic programming.
- We add concurrency to the second language. This gives a form of dataflow programming that is both purely functional and concurrent. We call it declarative concurrency. It is as easy to reason in as sequential functional programming. Eager execution and lazy execution are the two complementary ways to use declarative concurrency.
- We add explicit state to the second language. State is essential for modularity, because it allows changing a component's behavior over time without changing its interface. Object-oriented programming is a way of programming with state that adds concepts from knowledge representation, such as subtyping, class hierarchies, and associations. This leads to new program structuring techniques such as inheritance, delegation, and forwarding. All the techniques of the first language can be transferred to object-oriented programming.
- Using both concurrency and state together gives a language that is very expressive but also hard to program and reason in. There are two main approaches to master its complexity: using coarse-grained atomic actions such as monitors, or using message passing between active objects.

Within these languages and others, we discuss different forms of abstraction, nondeterminism, encapsulation, compositionality, security, and other important concepts. We give a simple operational semantics for all kernel languages. We show how some of the kernel languages can use other semantics such as axiomatic and logical semantics.

The kernel language approach is an outgrowth of ten years of programming language research and implementation by an international team, the Mozart Consortium, which consists of the Swedish Institute of Computer Science and the Royal Institute of Technology (KTH) in Sweden, the Universität des Saarlandes in Germany, and the Université Catholique de Louvain (UCL) in Belgium.

3. TEACHING EXPERIENCE

We first explain how we have realized the kernel language approach and our teaching experience with it. Based on this experience, we give recommendations on how to use the approach in an informatics curriculum.

3.1 Realization

We are writing a textbook *Concepts, Techniques, and Models of Computer Programming*. The latest draft is always available at:

http://www.info.ucl.ac.be/people/PVR/book.html

This draft is updated frequently and currently has more than 800 pages of material. It is intended for different levels of sophistication, ranging from second-year undergraduate courses to graduate courses. It assumes a previous exposure to programming and knowledge of simple mathematical concepts such as sequences and graphs. We have also prepared slides and lab sessions.

The textbook is supported by the Mozart Programming System, a full-featured open-source development platform that can run all program fragments in the book [26]. Full information including sources and binaries is available at the Mozart Web site:

http://www.mozart-oz.org

Mozart was originally developed as a vehicle for research in language design and implementation. We chose Mozart for the textbook because it implements the Oz language, which supports the kernel language approach perfectly well. Other reasons for picking Mozart are its implementation quality and its support for both Unix and Windows platforms.

The textbook mentions many languages and gives an in-depth treatment of four languages that are representative of widely different paradigms, namely Erlang, Haskell, Java, and Prolog. In a few pages it gives the essentials of each with respect to the kernel language approach and it gives the formal semantics of particularly interesting features.

3.2 Courses taught

The book draft and accompanying materials were used so far at three universities for the following courses:

- *DatalogiII 2G1512* (*Computer science II*, Fall 2001, 90 students, instructor Seif Haridi). Royal Institute of Technology (KTH), Kista, Sweden. For second-year students including both CS majors and non CS majors.
- *INGI2650* (*Structure of algorithmic programming languages*, Fall 2001, 55 students, instructor Peter Van Roy). Université catholique de Louvain (UCL), Louvain-la-Neuve, Belgium. For third-year CS students.

- *LINF1251* (*Introduction to programming part 2*, Spring 2002, 27 students, instructor Peter Van Roy). UCL. For second-year CS students. This follows a first-year introductory course based on a subset of Java.
- *INGI2655* (*Syntax and semantics of programming languages*, Spring 2002, 44 students, instructor Peter Van Roy). UCL. For fourth-year CS students. The book's operational semantics was used as a realistic example.
- *2G1915* (*Concurrent programming*, Spring 2002, 70 students, instructor Vladimir Vlassov). KTH. For fourth-year CS students. The chapter on concurrency and state was used.
- *EE 490/590* (Electrical and Computer Engineering Special Topics, instructor Juris Reinfelds). New Mexico State University, Las Cruces, New Mexico. Two CS graduate courses: *Distributed computing* (Fall 2001, 4 students) and *A programmer's theory of programming* (Spring 2002, 5 students). These courses and their motivation are covered in [30].

DatalogiII and INGI2650 were taught concurrently; they were the first time that the book was used for teaching. LINF1251 was the second time. Just for information, 64% passed DatalogiII,[1] 85% passed INGI2650, and 96% passed LINF1251. DatalogiII tried to teach too much material and the students (non CS majors) were less motivated. LINF1251 was more pedagogical: we adjusted the pace and all lectures were accompanied with live demonstrations and student interaction.

3.3 Curriculum recommendations

We have discussed the effects of the kernel language approach on the informatics curriculum with our colleagues at UCL, at workshops and conferences where we presented the approach, notably WCCE 2001 [2], MPOOL 2001 [11], and WFLP 2001 [17], and with other universities (notably the Katholieke Universiteit Leuven in Louvain, Belgium). Based on these discussions, we propose the following natural division of the discipline of programming into three core topics:

1. Concepts and techniques.
2. Algorithms and data structures.
3. Program design and software engineering.

These topics are focused on the discipline of programming, independent of any other domain in informatics. Our textbook gives a thorough treatment of topic (1) and an introduction to (2) and (3). Parnas presents an approach that focuses on topic (3) [28] After discussion, we agree that a good approach is

[1] This percentage does not count students who will retake the exam in the future.

to teach (1) and (3) at the same time, introducing concepts and design principles concurrently [29]. In the informatics curriculum at UCL, we attribute eight semester-hours to each topic. This includes both lectures and lab sessions. Together the three topics comprise one sixth of the complete informatics curriculum.

4. CONCLUSIONS AND FURTHER READING

We have given a brief overview and motivation of the kernel language approach to teaching programming. The approach focuses on programming concepts and the techniques to use them ("concepts first"), not on programming languages or paradigms. Practical languages are translated into closely related kernel languages, simple languages that present the essential concepts in an intuitive and precise way. This gives students a view that is both broad and deep. The approach covers many programming paradigms and shows their deep relationships. It has a simple formal semantics that is usable by practicing programmers.

For further reading there is an extensive overview talk that introduces the kernel languages and their semantics and gives two highlights, in concurrent programming and graphic user interface programming [33]. We also recommend reading the Preface and Appendix E (General Computation Model) of the draft textbook.

5. ACKNOWLEDGMENTS

We thank our teaching assistants Raphaël Collet, Frej Drejhammer, Sameh El-Ansary, and Dragan Havelka. We also thank Juris Reinfelds, Dave Parnas, Elie Milgrom, and Yves Willems. Finally, we thank the members of the Mozart Consortium. This research is partly financed by the Walloon Region of Belgium in the PIRATES project and the Belgian Fonds National de la Recherche Scientifique (FNRS).

6. REFERENCES

[1] Abelson, H., Sussman, G. J., and Sussman, J. (1996). *Structure and Interpretation of Computer Programs, Second Edition*. The MIT Press.

[2] Andersen, J. and Mohr, C., editors (2001). *Seventh IFIP World Conference on Computers in Education*. UNI-C Denmark.

[3] Andrews, G. R. (1991). Concurrent Programming: Principles and Practice. Addison-Wesley.

[4] Armstrong, J., Williams, M., Wikström, C., and Virding, R. (1996). Concurrent Programming in Erlang. Prentice Hall.
[5] Arnold, K. and Gosling, J. (1998). The Java Programming Language, Second Edition. Addison-Wesley.
[6] Bird, R. (1998). Introduction to Functional Programming using Haskell, Second Edition. Prentice Hall.
[7] Bratko, I. (2001). Prolog Programming for Artificial Intelligence, Third Edition. Addison-Wesley.
[8] Budd, T. (1995). Multiparadigm Programming in Leda. Addison-Wesley.
[9] Chailloux, E., Manoury, P., and Pagano, B. (2000). Développement d'Applications avec Objective Caml. O'Reilly, Paris, France.
[10] Cousineau, G. and Mauny, M. (1998). The Functional Approach to Programming. Cambridge University Press.
[11] Davis, K., Smaragdakis, Y., and Striegnitz, J., editors (2001). Workshop on Multiparadigm Programming with Object-Oriented Languages (at ECOOP 2001), volume 7. John von Neumann Institute for Computing.
[12] Dijkstra, E. W. (1997). A Discipline of Programming. Prentice Hall. Original publication in 1976.
[13] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2001). How to Design Programs: An Introduction to Computing and Programming. The MIT Press.
[14] Friedman, D. P., Wand, M., and Haynes, C. T. (1992). Essentials of Programming Languages. The MIT Press.
[15] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
[16] Hailperin, M., Kaiser, B., and Knight, K. (1999). Concrete Abstractions: An Introduction to Computer Science Using Scheme. Brooks/Cole Publishing Company.
[17] Hanus, M., editor (2001). International Workshop on Functional and (Constraint) Logic Programming. Christian-Albrechts-Universität Kiel. Bericht Nr. 2017.
[18] Hudak, P. (2000). The Haskell School of Expression: Learning Functional Programming Through Multimedia. Cambridge University Press.
[19] Kirkerud, B. (1997). Programming Language Semantics: Imperative and Object Oriented Languages. International Thomson Computer Press.
[20] Lea, D. (2000). Concurrent Programming in Java, Second Edition. Addison-Wesley.
[21] Liskov, B. and Guttag, J. (2000). Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley.
[22] MacLennan, B. J. (1987). Principles of Programming Languages, Second Edition. Saunders, Harcourt Brace Jovanovich.
[23] MacLennan, B. J. (1990). Functional Programming: Practice and Theory. Addison-Wesley.
[24] Main, M. (1999). Data Structures & Other Objects using Java. Addison-Wesley.
[25] Meyer, B. (2000). Object-Oriented Software Construction, Second Edition. Prentice Hall.
[26] Mozart Consortium (2001). The Mozart Programming System, Version 1.2.3. See http://www.mozart-oz.org.
[27] Nielson, H. R. and Nielson, F. (1992). Semantics with Applications: A Formal Introduction. John Wiley & Sons.
[28] Parnas, D. (1995). Teaching Programming as Engineering. Ninth International Conference of Z Users (Springer LNCS 967). Reprinted in Software Fundamentals, Addison-Wesley, 2001.
[29] Parnas, D. (2002). Private communication.

[30] Reinfelds, J. (2002). Teaching of Programming with a Programmer's Theory of Programming. ICTEM 2002, Kluwer Academic Publishers.

[31] Sterling, L. and Shapiro, E. (1994). The Art of Prolog: Advanced Programming Techniques, Second Edition. Series in Logic Programming. The MIT Press.

[32] Stroustrup, B. (1997). The C++ Programming Language, Third Edition. Addison-Wesley.

[33] Van Roy, P. and Haridi, S. (2002). Teaching Programming both Broadly and Deeply: The Kernel Language Approach. Talks. See http://www.info.ucl.ac.be/people/PVR/book.html.

[34] Winskel, G. (1993). The Formal Semantics of Programming Languages. Foundations of Computing Series. The MIT Press.