Teaching Programming with the Kernel Language Approach

Peter Van Roy[1] and Seif Haridi[2]

[1] Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
pvr@info.ucl.ac.be
[2] Royal Institute of Technology (KTH), S-164 40 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) in 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. We have taught the approach at three universities in courses ranging from second-year undergraduate courses to graduate courses. We are completing a textbook with accompanying materials. As part of a curriculum, the approach naturally complements courses on algorithms & data structures and program design & software engineering. The approach is the fruit of ten years of research by an international team, the Mozart Consortium.

1 Existing Approaches

For the purposes of this paper, let us consider a broad definition of *computer programming* as bridging the gap between specification and running program. This consists in designing the architecture and abstractions of an application and coding them in a programming language. The discipline of programming has two essential parts: a technology and its scientific foundation. The technology consists of tools, techniques, and standards, allowing to *do* programming. The science consists of a broad and deep theory with predictive power, allowing to *understand* programming. The science should be practical, that is, able to explain the technology, making it useful for a practicing programmer.

Teaching programming means to teach both the science and the technology. Surprisingly, we find that programming is almost never taught in this way. Surveying existing textbooks, we find that programming is taught in three different ways.

1.1 As a Craft

The most popular approach is to teach programming as a craft in the context of a single programming paradigm, embodied in a single language. The science is limited to the chosen paradigm or language. Some popular paradigms are object-oriented programming [23, 26, 27, 16], imperative programming [24], functional programming [11, 19, 7, 14, 25], logic programming [33, 8], and concurrent imperative programming [4, 5]. Only the textbooks on functional programming and concurrent imperative programming give a formal semantics. Some languages are Java [23, 26, 22, 6], C++ [34], Eiffel [27], Prolog [33, 8], Erlang [5], Objective Caml [10], and Leda [9]. Leda is presented as a multiparadigm language but it contains only a few paradigms and these are presented in isolation.

Teaching programming in terms of a single paradigm or language 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 [22]. 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.

1.2 As a Branch of Mathematics

The second approach is to teach programming as a branch of mathematics. In this approach, the science is either limited to a restricted language, as in [13], or too fundamental to be of practical use, as in [37, 20, 29].

The approach given by Dijkstra in [13] is practical but it only treats a restricted set of concepts. A promising continuation of [13] would be to extend it to more concepts. This is part of our approach, which is explained in Section 2.

1.3 In Terms of Concepts

The third approach is to teach programming in terms of the underlying concepts. The concepts are elaborated starting from simple ones and gradually introducing more sophisticated ones. This gives students a good understanding of practical programming. For example, object-oriented programming is explained in terms of functional programming by adding explicit state. Textbooks that use this approach are Abelson *et al* and its successors [1, 2, 15, 17]. However, these textbooks lack formal semantics and they leave out many important concepts.

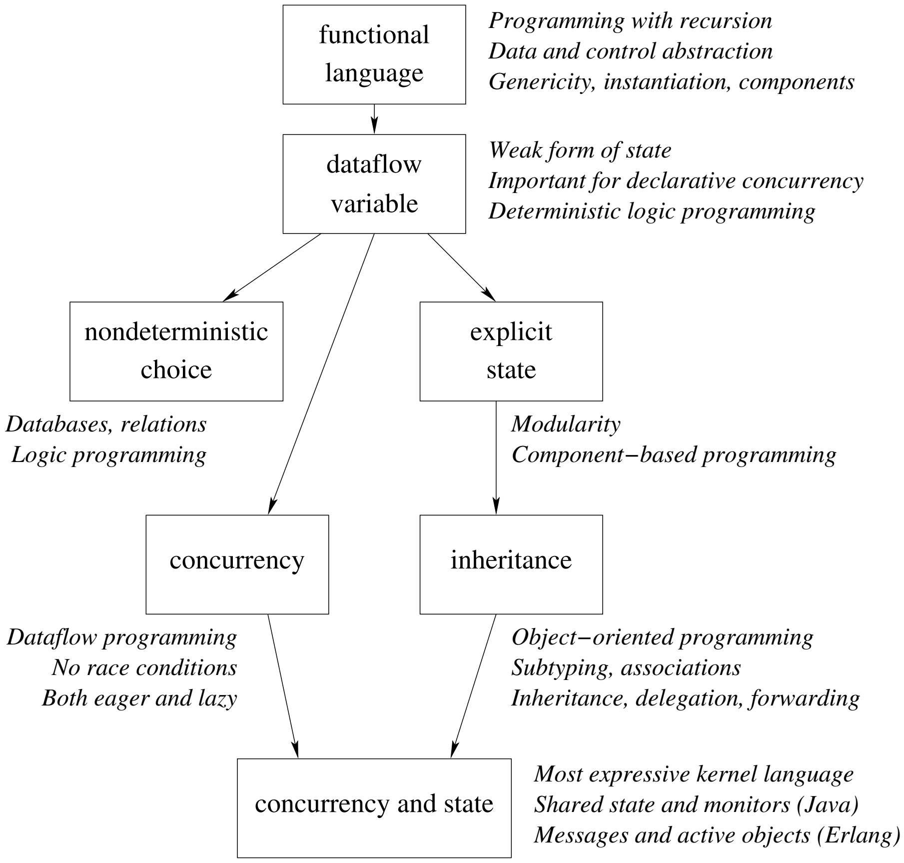

Fig. 1. Some important steps in the kernel language approach.

Our approach is also based on concepts. We extend Abelson *et al* by treating more concepts, by putting the treatment of concepts on a better methodological foundation, and by adding a formal semantics.

2 The Kernel Language Approach

How can we teach programming as a unified discipline? 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 and instructor (since each language has its own syntax and semantics) but does not show the deep connections between the paradigms.

A better approach would not be based on a single language (or a few languages), but on the underlying concepts. This is the approach taken by Abelson *et al* [1] and its successors, which we mentioned in the previous section. We have extended this approach to be both broader and deeper than has been done before. We organize the concepts 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 modeled 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. We give an operational semantics for all kernel languages and axiomatic and logical semantics for those kernel languages for which it is appropriate. This is in contrast to [1, 2, 15], which define languages in terms of interpreters. Compared to giving a simple operational semantics, we find that using interpreters is obscure (since language features interact in nonobvious ways) and makes it harder to reason abstractly about program correctness and complexity. Our operational semantics is carefully factored by concept. This makes the semantics easy to understand and makes it easy to define a single operational semantics that can be subsetted for many different kernel languages.

In the kernel language approach, programming paradigms and their requisite languages are 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.

2.1 The Kernel Languages

Figure 1 briefly summarizes how we organize programming according to this approach. Each box corresponds to a kernel language. The first box contains a simple functional language and successive boxes add concepts incrementally. 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, which are a kind of single-assignment variable, i.e., a weak form of state. (State is such a strong addition that it is useful to have a weaker form of it.) This has two important consequences. 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 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 can be seen as a rich set of programming techniques with state, centered around encapsulation (programs as collections of abstract data types) and subtyping (the incremental definition of abstract data types). It emphasizes structuring techniques such as inheritance, delegation, and forwarding.
– 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 [21]. One approach is to use atomic actions on shared state, such as monitors in Java [22]. Another approach is to use message passing between active objects, as in Erlang [5].

Within these languages and others we discuss different forms of abstraction, nondeterminism, encapsulation, compositionality, capability-based programming, and other important concepts.

2.2 The Creative Extension Principle

We organize the kernel languages in a layered fashion, starting from a simple base language and successively adding new concepts. How do we decide which concepts to add? Whenever programming in a kernel language becomes complex for technical reasons independent of the program, this is a signal that there is a new concept waiting in the wings. If the concept is chosen well, then adding it to the language satisfies two properties: programs remain simple and the language semantics also remains simple. Here are some examples:

– In the first kernel language, programming independent activities requires programming a scheduler and explicit context switches. Adding concurrency to the language avoids this complexity. Each activity can be programmed as if it was the sole activity, and it needs to know about other activities only when interacting with them.
– In the first kernel language, a procedure cannot change its behavior over time without affecting its interface. Adding explicit state to the language avoids this complexity.
– All the other concepts are added by following the same approach. For example, we add dataflow, exception handling, lazy execution, capabilities, and nondeterministic choice. In each case, encoding the concept without changing the language increases the complexity of the program. In each case, adding the concept to the kernel language keeps both the program and the language semantics simple.

This approach has a strong element of creativity. Each concept brings something novel that was not there before. We therefore call it the *creative extension principle*. It is the foundation of how we organize the kernel languages. The organization we propose is not the only possible one, however. By following the same approach, you may find a different organization. We would be interested to exchange ideas with anyone who has done this.

3 Teaching Experience

We first explain how we have realized the kernel language approach for educational purposes 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 Textbook and Software

We have realized the kernel language approach in the textbook *Concepts, Techniques, and Models of Computer Programming*, which is currently being completed. The latest draft is always available at the URL cited in the bibliography [35]. 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 basic knowledge of simple mathematical concepts such as sequences, graphs, and functions. We have also prepared slides and lab sessions, which we offer to interested parties on request.

The textbook does not target first-year students, although the approach could probably be adapted for such a purpose. We would be interested in contacting anyone attempting such an adaptation.

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 [28]. Full information including downloadable sources and binaries is available at the Mozart Web site. 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. Of course, the kernel language approach could be supported by other languages and software platforms. We encourage efforts in this direction and we would be happy to exchange ideas with anyone undertaking such an effort.

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 have so far been used at four universities for the following courses:

- *DatalogiII 2G1512* (*Computer science II*, Fall 2001 and Fall 2002, 90 students in Fall 2001, instructors Seif Haridi and Christian Schulte). 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 course, instructor Juris Reinfelds). New Mexico State University, Las Cruces, New Mexico. Two 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 [32].
- *CS437* (*Distributed systems*, Spring 2001, 45 students, instructor Reem Bahgat). Cairo University, Cairo, Egypt. For fourth-year CS students.
- *CS532* (*Declarative programming systems*, Fall 2001, 16 students, instructor Reem Bahgat). Cairo University. For master's students in CS (fifth year).

Feedback from these courses was used to improve the book's content and organization. DatalogiII and INGI2650 in Fall 2001 were the first major uses of the book for teaching. LINF1251 was the second major use. Just for information, 64% passed the first instance of DatalogiII,[3] 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.

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 [3], MPOOL 2001 [12], WFPL 2001 [18], and ICTEM 2002 [36], and with other universities (notably the Katholieke

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

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) [30]. After discussion with Parnas, we agree that a good approach is to teach (1) and (3) concurrently, introducing both concepts and design principles gradually [31]. 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 undergraduate informatics curriculum at UCL.

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, 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 the deep relationships between them. It has a simple formal semantics that is usable by practicing programmers. It extends similar approaches, e.g., by Abelson *et al* [2], with more concepts, a better methodological foundation, and formal semantics.

For further reading we recommend the overview talk, which introduces the kernel languages and their semantics and gives two highlights, in concurrent programming and graphic user interface programming [36]. We also recommend reading the Preface and Appendix E (General Computation Model) of the draft textbook. Appendix E is especially relevant: it explains the creative extension principle, which is the foundation we use to determine which concepts to put in the kernel languages and how to classify the kernel languages.

Acknowledgments

The kernel language approach is an outgrowth of ten years in 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.

An early version of this paper appeared in ICTEM 2002 in July 2002 [36]. 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. We thank an anonymous reviewer for his useful comments that helped improve the paper's presentation. This research is partly financed by the Walloon Region of Belgium in the PIRATES project.

References

1. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Mass, 1985.
2. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. The MIT Press, Cambridge, Mass, 1996.
3. Jane Andersen and Christine Mohr, editors. *Seventh IFIP World Conference on Computers in Education*. UNI-C Denmark, 2001.
4. Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.
5. J. Armstrong, M. Williams, C. Wikström, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, N.J., 1996.
6. Ken Arnold and James Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, 1998.
7. Richard Bird. *Introduction to Functional Programming using Haskell, Second Edition*. Prentice Hall, 1998.
8. Ivan Bratko. *Prolog programming for artificial intelligence, Third Edition*. Addison-Wesley, 2001.
9. Timothy A. Budd. *Multiparadigm Programming in Leda*. Addison-Wesley, 1995.
10. Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, Paris, France, 2000.
11. Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
12. Kei Davis, Yannis Smaragdakis, and Jörg Striegnitz, editors. *Workshop on multiparadigm programming with object-oriented languages (at ECOOP 2001)*, volume 7. John von Neumann Institute for Computing, 2001.
13. Edsger W. Dijkstra. *A discipline of programming*. Prentice Hall, 1997. Original publication in 1976.
14. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Computing and Programming*. The MIT Press, 2001.
15. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, 1992.
16. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
17. Max Hailperin, Barbara Kaiser, and Karl Knight. *Concrete Abstractions: An Introduction to Computer Science Using Scheme*. Brooks/Cole Publishing Company, 1999.
18. Michael Hanus, editor. *International Workshop on Functional and (Constraint) Logic Programming*. Christian-Albrechts-Universität Kiel, 2001. Bericht Nr. 2017.

19. Paul Hudak. *The Haskell School of Expression: Learning Functional Programming Through Multimedia.* Cambridge University Press, 2000.
20. Bjørn Kirkerud. *Programming Language Semantics: Imperative and Object Oriented Languages.* International Thomson Computer Press, 1997.
21. Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. In *Second International Symposium on Operating Systems, IRIA*, October 1978. Reprinted in *Operating Systems Review*, 13(2), April 1979, pp. 3–19.
22. Doug Lea. *Concurrent Programming in Java, Second Edition.* Addison-Wesley, 2000.
23. Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley, 2000.
24. Bruce J. MacLennan. *Principles of Programming Languages, Second Edition.* Saunders, Harcourt Brace Jovanovich, 1987.
25. Bruce J. MacLennan. *Functional Programming: Practice and Theory.* Addison-Wesley, 1990.
26. Michael Main. *Data Structures & Other Objects Using Java.* Addison-Wesley, 1999.
27. Bertrand Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice Hall, 2000.
28. Mozart Consortium. The Mozart Programming System version 1.2.3, December 2001. Available at `http://www.mozart-oz.org/`.
29. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* John Wiley & Sons, 1992.
30. Dave Parnas. Teaching programming as engineering. In *9th International Conference of Z Users.* Springer-Verlag, 1995. Lecture Notes in Computer Science, vol. 967. Reprinted in *Software Fundamentals*, Addison-Wesley, 2001.
31. Dave Parnas. Private communication, 2002.
32. Juris Reinfelds. Teaching of programming with a programmer's theory of programming. In *Informatics Curricula, Teaching Methods, and Best Practice (ICTEM 2002).* Kluwer Academic Publishers, 2002.
33. Leon Sterling and Ehud Shapiro. *The Art of Prolog–Advanced Programming Techniques.* Series in Logic Programming. The MIT Press, 1986.
34. Bjarne Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.
35. Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* 2002. Draft available at `http://www.info.ucl.ac.be/people/PVR/book.html`.
36. Peter Van Roy and Seif Haridi. Teaching programming broadly and deeply: the kernel language approach. In *Informatics Curricula, Teaching Methods, and Best Practice (ICTEM 2002).* Kluwer Academic Publishers, 2002. Also two talks (short and long), available at `http://www.info.ucl.ac.be/people/PVR/book.html`.
37. Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing Series. The MIT Press, Cambridge, Massachusetts, 1993.