Teaching Programming with the Kernel Language Approach

October 7, 2002

Functional and Declarative Programming in Education (FDPE02)
Workshop at PLI 2002

Peter Van Roy
Université catholique de Louvain (UCL)
Louvain-la-Neuve, Belgium

Seif Haridi
Royal Institute of Technology (KTH)
Kista, Sweden

© 2002 P. Van Roy and S. Haridi
Overview

• Programming needs both technology and science
  – Current approaches to teach programming are lacking

• Example: concurrent programming
  – Monitors in Java
  – The broad view

• The kernel language approach
  – A family of kernel languages
  – Formal semantics for the practicing programmer
  – Creative extension principle

• Teaching experience
  – Textbook and software
  – Courses taught
  – Curriculum recommendations

• Conclusions
What is programming?

• We define **programming** broadly as the step from specification to running program, which consists in designing the architecture and its abstractions and coding them into a programming language.

• Doing programming well requires understanding two topics:
  – A **technology**: a set of practical techniques, tools, and standards
  – A **science**: a scientific theory that explains the technology

• Teaching programming well therefore requires teaching both the technology and the science:
  – Surprisingly, programming is almost never taught in this way. It is almost always taught as a **craft** in the context of current technology (e.g., Java and its tools). If there is any science, it is either limited to the tools or too theoretical.

• We propose a remedy, **the kernel language approach**
Concurrent programming: monitors in Java

• Concurrent programming with shared state and monitors (as done in Java) is **so complicated** that it is taught only in advanced courses (upper level undergraduate)
• The implementation of concurrency in Java is **expensive**
• Java-taught programmers therefore reach the conclusion that concurrency is **always complicated and expensive**
• But this is **completely false**: there are useful forms of concurrency (e.g., dataflow, streams, active objects) that are easy to use and can be implemented efficiently
• Therefore programmers should be taught about concurrency in a broader way
Concurrent programming: the broad view

• We distinguish four forms of practical concurrent programming (in order of increasing difficulty):
  – Sequential programming + variants
  – Declarative concurrency (lazy and eager): add threads to a functional language and use dataflow to decouple independent calculations
  – Message passing between active objects: Erlang style, each thread runs a functional program, threads communicate through asynchronous channels
  – Atomic actions on shared state: Java style, using monitors and transactions

• The Java style is the most popular, yet it is the most difficult to program
• Declarative concurrency especially is quite useful, yet is not widely known
  – Programming with streams and dataflow
  – All the programming and reasoning techniques of sequential declarative programming apply (concurrent programs give the same results as sequential ones)
  – Deep characterization: lack of observable nondeterminism
Approaches to teach programming

• As a craft
  – Most popular; single paradigm and language

• As a branch of mathematics
  – Usually too theoretical
  – Dijkstra has done this successfully, but with only a small language

• In terms of concepts
  – Start with simple concepts and gradually introduce more sophisticated ones, as they are needed
  – The concepts are not limited to single languages or paradigms
  – Abelson & Sussman and its successors use this approach
The kernel language approach

• How can we teach programming as a unified discipline?
  – There are too many languages
  – Teaching a few carefully-selected languages, say one per paradigm, does not solve
    the problem: it multiplies the effort of student and teacher but does not show the
    deep relationships between the paradigms

• A better approach would be based on concepts, not languages, as done by
  Abelson & Sussman

• We organize the concepts into simple languages called kernel languages
  – A wide variety of languages and programming paradigms can be translated into a
    small set of closely-related kernel languages
  – We give an operational semantics in terms of a simple abstract machine at a high
    level of abstraction
  – We try to be as comprehensive as possible, incorporating all of the most important
    concepts. In particular, we have a comprehensive treatment of concurrency.
  – We organize the concepts according to the creative extension principle
Related work

• By far the closest books are “Structure and Interpretation of Computer Programs”, by Abelson & Sussman, and its successor “Essentials of Programming Languages”, by Friedman et al.
  – Both these books and ours are based on concepts: they “liberate programming from the tyranny of syntax” (Felleisen et al)

• Our approach differs in four major ways:
  – Translation:
    • We translate into kernel languages instead of writing interpreters
  – Formal semantics:
    • We give a simple but precise abstract machine that allows reasoning about time and space complexity.
  – Breadth:
    • We go deeper into concurrency, capabilities, and logic programming. We apply the approach to user interfaces, distributed computing, and constraint programming. All concepts are fully implemented in the Mozart system.
  – Methodology:
    • We organize the concepts according to the creative extension principle, which indicates when new concepts are needed and gives a criterium for judging them
The kernel language approach (2)

- Kernel languages have a small number of programmer-significant elements
- Their purpose is to understand programming from the programmer’s viewpoint
- They are given a semantics which allows the practicing programmer to reason about correctness and complexity at a high level of abstraction
The kernel language approach (3): analogy with classical mechanics

- Classical mechanics is a branch of physics that is widely used in engineering
- Classical mechanics is based on a small set of physical laws
- These laws can be formulated in three basically different ways, which are useful for different communities
- For engineers, the formulation based on Newton’s laws (and its derivations) is the most useful in practice (back of envelope)
What concepts should be in the kernel languages?

• There are many possibilities
  – We propose a methodology to design kernel languages
  – The methodology underlies our textbook and pedagogy

• Creative extension principle
  – Start from a simple base language
  – Programming with this language exposes limitations in expressiveness
    • Programs become complex for reasons independent of the application
    • This means that there is a new concept waiting in the wings!
    • Examples: exceptions, capabilities, concurrency, laziness, search, state
  – There is always a choice:
    • To encode the concept in the language, which makes programs complicated but
      keeps the language semantics simple
    • To add the concept to the language. If the concept is chosen well, the program
      becomes simple and the language semantics is extended in a modular way.
      – Can always program in the original subset to get original semantics back
  – Iterating this process gives a family of kernel languages
A family of kernel languages

Declarative model
- strict functional programming, e.g., *Scheme*
- deterministic logic programming
  + dataflow concurrency
  + by-need synchronization
  - declarative concurrency
- lazy functional programming, e.g., *Haskell*
  + nondeterministic choice
  - concurrent logic programming
  + explicit state
  + exception handling
  - object-oriented programming
    + encapsulated search
      - nondeterministic LP, e.g., *Prolog*
- concurrent OOP
  (active object style, e.g., *Erlang*)
  (shared state style, e.g., *Java*)
  + monotonic assignment
  - constraint programming

- The kernel languages are closely related
- Each kernel language has its own reasoning techniques and its own programming techniques
- These techniques can also be used in extended kernel languages
- There are many more kernel languages than are listed here

© 2002 P. Van Roy and S. Haridi
**Most general language (so far)**

\[ \langle s \rangle ::= \]

- `skip`
- `\langle s \rangle_1 \langle s \rangle_2`
- `local \langle x \rangle in \langle s \rangle end`
- `\langle x \rangle_1 = \langle x \rangle_2`
- `\langle x \rangle = \langle v \rangle`

\{ `\langle x \rangle \langle y \rangle_1 \ldots \langle y \rangle_n`\}

- `if \langle x \rangle then \langle s \rangle_1 else \langle s \rangle_2 end`
- `case \langle x \rangle of \langle p \rangle then \langle s \rangle_1 else \langle s \rangle_2 end`
- `thread \langle s \rangle end`
- `{ByNeed \langle x \rangle_1 \langle x \rangle_2}`

\( (\text{choice} + \text{search}) \)

- `{NewName \langle x \rangle}`
- `try \langle s \rangle_1 catch \langle x \rangle then \langle s \rangle_2 end`
- `raise \langle x \rangle end`
- `{NewCell \langle x \rangle_1 \langle x \rangle_2}`
- `{Exchange \langle x \rangle_1 \langle x \rangle_2 \langle x \rangle_3}`

**Empty statement**
- Statement sequence
- Variable creation
- Variable-variable binding
- Value creation

**Procedure application**
- Conditional
- Pattern matching
- Thread creation
- Trigger creation (laziness)

**Encapsulated search**

**Name creation (security)**
- Exception context
- Raise exception
- Cell creation
- Cell exchange

© 2002 P. Van Roy and S. Haridi
Most general language (2)

- There are three kinds of values in the language:
  numbers, records, and procedures

\[
\text{<v> ::= <number> | <record> | <procedure>}
\]

\[
\text{<number> ::= <int> | <float>}
\]

\[
\text{<record>, <p> ::= <lit>(<feat>}_1:<x}_{1} \ldots \text{ <feat>}_n:<x}_{n})
\]

\[
\text{<procedure> ::= proc \{ <x>}_1 \ldots <x>_n \} \ <s> \ end}
\]

\[
\text{<lit> ::= <atom> | <bool>}
\]
\[
\text{<feat> ::= <atom> | <bool> | <int>}
\]
\[
\text{<bool> ::= true | false}
\]
Formal semantics (1)

- We define a simple but precise abstract machine
  - Other semantics tie on to this (SOS, axiomatic, logical)
- Basic concepts:
  - A *single-assignment store* $\sigma$ is a set of store variables $x_1, \ldots, x_k$, that are partitioned into sets of equal unbound variables and variables bound to a number, record, or procedure
  - An *environment* $E$ is a mapping from variable identifiers to store variables, $\{<x>_1 \rightarrow x_1, \ldots, <x>_n \rightarrow x_n\}$
  - A *semantic statement* is a pair $(<s>, E)$ where $<s>$ is a statement and $E$ is an environment
  - An *execution state* is a pair $(ST, \sigma)$ where $ST$ is a stack of semantic statements
  - A *computation* is a sequence of execution states starting from an initial state: $(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow \ldots$
Formal semantics (2)

- Program execution
  - The initial execution state is $[(<s>,\phi)], \phi)$. The initial semantic statement is $(<s>,\phi)$ with an empty environment, and the initial store is empty.
  - At each execution step, the first element of $ST$ is popped and execution proceeds according to the form of the element.
  - The final execution state (if it exists) is one in which the semantic stack is empty.

- A semantic stack can be in one of three run-time states:
  - running: $ST$ can do an execution step
  - terminated: $ST$ is empty
  - suspended: $ST$ is not empty but cannot do a step
Example: the \textbf{local} statement

- The semantic statement is (\textbf{local} \textit{<x> in <s> end, E})

- Execution consists of the following actions:
  - Create a new variable \textit{x} in the store
  - Push (\textit{<s>, E+{<x>→x}}) on the stack

- Students clearly see the difference between \textbf{identifiers} (bits of syntax, like \textit{<x>}) and \textbf{variables in memory} (entities that take part in the computation, like \textit{x})
Example: the if statement

- The semantic statement is \( \text{if } <x> \text{ then } <s> \_1 \text{ else } <s> \_2 \text{ end, } E \)\)

- This statement has an activation condition: \( E(<x>) \) must be determined, i.e., bound to a number, record, or procedure

- Execution consists of the following actions:
  - If the activation condition is true, then do the following actions:
    - If \( E(<x>) \) is not a boolean (true or false), then raise an error condition
    - If \( E(<x>) \) is true, then push \( (<s> \_1, E) \) on the stack
    - If \( E(<x>) \) is false, then push \( (<s> \_2, E) \) on the stack
  - If the activation condition is false, then execution suspends

- If some other activity in the system makes the activation condition true, then execution can continue. This does dataflow programming, which is at the heart of declarative concurrency.
Example: procedures

- A procedure value is a pair \(\texttt{proc} \{ \langle y\rangle_1 \ldots \langle y\rangle_n \} \langle s\rangle \texttt{end, CE}\) where \(CE\) (the « contextual environment ») is \(E|_{\langle z\rangle_1, \ldots, \langle z\rangle_m}\), where \(E\) is the environment where the procedure is defined and \(\{\langle z\rangle_1, \ldots, \langle z\rangle_m\}\) is the set of external identifiers of the procedure.

- In a procedure call \(\{\langle x\rangle \langle x\rangle_1 \ldots \langle x\rangle_n\}, E\):
  - if \(E(\langle x\rangle)\) has the form \(\texttt{proc} \{ \langle y\rangle_1 \ldots \langle y\rangle_n \} \langle s\rangle \texttt{end, CE}\), then
  - push \(\langle s\rangle, CE + \{ \langle y\rangle_1 \mapsto E(\langle x\rangle_1), \ldots, \langle y\rangle_n \mapsto E(\langle x\rangle_n)\}\)

- This allows higher-order programming as in functional languages
  - A basic building block for abstraction, genericity, instantiation, and embedding, the techniques that underlie objects and components.
Programming paradigms as epiphenomena

• The kernel approach lets us organize programming in three levels:
  – **Concepts**: compositionality, encapsulation, lexical scoping, higher-orderness, capability property, concurrency, dataflow, laziness, state, inheritance, ...
  – **Techniques**: how to write programs with these concepts
  – **Computation models** (« paradigms »): each model contains a fixed set of concepts and is realized with data entities, operations, and a language

• Programming paradigms *emerge in a natural way* when programming (as a kind of epiphenomenon), depending on which concepts one uses in a model and which properties hold of the resulting model
  – **Reasoning techniques** depend on paradigm. Paradigms with fewer concepts are less expressive but simplify reasoning.

• It is often advantageous for programs to use several paradigms together (examples: concurrency, user interfaces, …)
Teaching experience

• Materials
    • See: http://www.info.ucl.ac.be/people/PVR/book.html
    • Work in progress since early 2000; recently sent to publisher
  – Software: Mozart Programming System
    • See: http://www.mozart-oz.org/
    • Open source system used in many R&D projects; active development since 1991
    • Implements the Oz language (fits well the kernel language approach)
    • Developed by the Mozart Consortium (groups in Germany, Sweden, Belgium)
  – Transparencies, lab sessions, interactive demos

• Courses taught (at UCL, KTH, NMSU, Cairo University)
  – Audiences covered so far: second to fourth year CS majors, graduate CS majors, second-year engineering (both CS and non CS majors)
  – Course topics: introduction to programming, algorithmic programming concepts, semantics, concurrent programming, distributed computing, declarative programming

• Not intended as a first course
  – The approach could likely be adapted; we have not done this
Curriculum recommendations

• We propose the following division of the discipline of programming into three topics:
  – Concepts and techniques
  – Algorithms and data structures
  – Program design and software engineering
• We recommend teaching the first and third topics together, introducing concepts and design principles concurrently
  – Textbook treats topic 1 in depth and gives introductions to the others
• At UCL, each topic is given 8 semester-hours (lectures + lab sessions)
  – All three together take one full semester, spread out over the complete curriculum
  – The complete curriculum has three full years of CS topics supplemented with one or two full years of non-CS topics for the licentiate and engineering degrees respectively
Conclusions

- The kernel language approach focuses on concepts and programming techniques, not on programming languages or paradigms
- Practical languages are translated into simple kernel languages based on small sets of programmer-significant concepts
  - The kernel languages have much in common, which allows them to show clearly the deep relationships between different languages and programming paradigms
  - We give a semantics at the right level of abstraction for the practicing programmer, to allow reasoning about correctness and complexity
- We support the approach with a textbook, teaching materials, and a software platform
  - We are teaching with the textbook in four universities (F 2001, Sp 2002, …), from second-year to graduate courses
  - The textbook extends the concepts-first approach of Abelson & Sussman with formal semantics, wider coverage, and a justifiable choice of concepts
  - The software platform is high quality and runs all programs in the book
- Based on our experience, we give recommendations on how to teach programming in the CS curriculum