Concepts, Techniques, and Models of Computer Programming

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

Seif Haridi
Kungliga Tekniska Högskolan
Kista, Sweden
Overview

- Goals of the book
  - What is programming?
- Concepts-based approach
  - History
  - Creative extension principle
- Teaching programming
- Examples to illustrate the approach
  - Concurrent programming
  - Data abstraction
  - Graphical user interface programming
  - Object-oriented programming: a small part of a big world
- Formal semantics
- Conclusion
Goals of the book

- To present programming as a unified discipline in which each programming paradigm has its part
- To teach programming without the limitations of particular languages and their historical accidents of syntax and semantics
- Today’s talk will touch on both of these goals
What is programming?

- Let us define “programming” broadly
  - The act of extending or changing a system’s functionality
  - For a software system, it is the activity that starts with a specification and leads to its solution as a program

- This definition covers a lot
  - It covers both programming “in the small” and “in the large”
  - It covers both (language-independent) architectural issues and (language-dependent) coding issues
  - It is unbiased by the limitations of any particular language, tool, or design methodology
Concepts-based approach

- Factorize programming languages into their primitive concepts
  - Depending on which concepts are used, the different programming paradigms appear as epiphenomena
  - Which concepts are the right ones? An important question that will lead us to the creative extension principle: add concepts to overcome limitations in expressiveness.
- For teaching, we start with a simple language with few concepts, and we add concepts one by one according to this principle
- We have applied this approach in a much broader and deeper way than has been done before
  - Using research results from a long-term collaboration
History (1)

- The concepts-based approach distills the results of a long-term research collaboration that started in the early 1990s
  - ACCLAIM project 1991-94: SICS, Saarland University, Digital PRL, …
    - AKL (SICS): unifies the concurrent and constraint strains of logic programming, thus realizing one vision of the FGCS
    - LIFE (Digital PRL): unifies logic and functional programming using logical entailment as a delaying operation (logic as a control flow mechanism!)
    - Oz (Saarland U): breaks with Horn clause tradition, is higher-order, factorizes and simplifies previous designs
  - After ACCLAIM, these partners decided to continue with Oz
  - Mozart Consortium since 1996: SICS, Saarland University, UCL
- The current design is Oz 3
  - Both simpler and more expressive than previous designs
  - Distribution support (transparency), constraint support (computation spaces), component-based programming
  - High-quality open source implementation: Mozart
History (2)

- In the summer of 1999, the two authors realized that they understood programming well enough to teach it in a unified way
  - We started work on a textbook and we started teaching with it
  - Little did we realize the amount of work it would take. The book was finally completed near the end of 2003 and turned out a great deal thicker than we anticipated.

- Much new understanding came with the writing and organization
  - The book is organized according to the creative extension principle
  - We were much helped by the factorized design of the Oz language; the book “deconstructs” this design and presents a large subset of it in a novel way

- We rediscovered important computer science that was “forgotten”, e.g., determinate concurrency, objects vs. ADTs
  - Both were already known in the 1970s, but largely ignored afterward!
Creative extension principle

- Language design driven by limitations in expressiveness
- With a given language, when programs start getting complicated for technical reasons unrelated to the problem being solved, then there is a new programming concept waiting to be discovered
  - Adding this concept to the language recovers simplicity
- A typical example is exceptions
  - If the language does not have them, all routines on the call path need to check and return error codes (non-local changes)
  - With exceptions, only the ends need to be changed (local changes)
- We rediscovered this principle when writing the book!
  - Defined formally and published in 1990 by Felleisen et al.
Example of creative extension principle

Language without exceptions

proc {P1 ... E1}
{P2 ... E2}
if E2 then ... end
E1=...
end

proc {P2 ... E2}
{P3 ... E3}
if E3 then ... end
E2=...
end

proc {P3 ... E3}
{P4 ... E4}
if E4 then ... end
E3=...
end

proc {P4 ... E4}
if (error) then E4=true
else E4=false end

Language with exceptions

try
{P2 ...}
catch E then ... end
end

proc {P2 ...}
{P3 ...}
end

proc {P3 ...}
{P4 ...}
end

proc {P4 ...}
if (error) then
raise myError end
end

Error occurs here

Error treated here

All procedures on path are modified

Error occurs here

Error treated here

Unchanged

Only procedures at ends are modified
Taxonomy of paradigms

Declarative programming

- Strict functional programming, Scheme, ML
- Deterministic logic programming, Prolog
  + concurrency
  + by-need synchronization
  Declarative (dataflow) concurrency
- Lazy functional programming, Haskell
  + nondeterministic choice
- Concurrent logic programming, FCP
  + exceptions
  + explicit state
- Object-oriented programming, Java, C++
  + search
- Nondeterministic logic prog., Prolog
  + computation spaces
  Constraint programming

- Concurrent OOP
  (message passing, Erlang, E)
  (shared state, Java)

- This diagram shows some of the important paradigms and how they relate according to the creative extension principle.
- Each paradigm has its pluses and minuses and areas in which it is best.
Complete set of concepts (so far)

| `<s>` ::= |
|---|---|
| `skip` | Empty statement |
| `<x>` `=` `<x>` | Variable binding |
| `<x>` `=` `<record>` | Value creation |
| `<x>` `=` `<number>` | Sequential composition |
| `<x>` `=` `<procedure>` | Variable creation |
| `local` `<x>` `in` `<s>` | |
| `end` | |
| `if` `<x>` `then` `<s>` `1` `else` `<s>` `2` `end` | Conditional |
| `case` `<x>` `of` `<p>` `then` `<s>` `1` `else` `<s>` `2` `end` | Pattern matching |
| `{<x> `<y>` `1` ... `<y>` `n`}` | Procedure invocation |
| `thread` `<s>` `end` | Thread creation |
| `{WaitNeeded `<x>`}` | By-need synchronization |
| `{NewName `<x>`}` | |
| `<x>` `=` `!!<x>` | |
| `try` `<s>` `1` `catch` `<x>` `then` `<s>` `2` `end` | Name creation |
| `raise` `<x>` `end` | Read-only view |
| `{NewPort `<x>` `1` `<x>` `2`}` | Exception context |
| `{Send `<x>` `1` `<x>` `2`}` | Raise exception |
| `<space>` | Port creation |
| | Port send |
| | Encapsulated search |
Complete set of concepts (so far)

### Syntax Rules

- **Expression** \( <s> ::= \)**
  - `skip`
  - \(<x>_1=<x>_2\)
  - \(<x>=<\text{record}> \mid <\text{number}> \mid <\text{procedure}>\)
  - \(<s>_1 <s>_2\)
  - `local <x> in <s> end`

- **Conditional**
  - `if <x> then <s>_1 else <s>_2 end`

- **Pattern matching**
  - `case <x> of <p> then <s>_1 else <s>_2 end`
  - `{<x> <y>_1 ... <y>_n}\`

- **Procedure invocation**
  - `thread <s> end`
  - `{\text{WaitNeeded} <x}>\`

- **Thread creation**
  - `{\text{Name creation} <x>\}
  - `<x>_1= !!<x>_2`

- **By-need synchronization**
  - `try <s>_1 catch <x> then <s>_2 end`
  - `raise <x> end`
  - `{\text{Name creation} <x> \mid <x>_2}`
  - `{\text{Exception context} <x>_1 <x>_2 <x>_3}`

- **Encapsulated search**
  - `<\text{Encapsulated search}>\`

---

9/12/2004  
P. Van Roy, BCS talk
Teaching programming

- How can we teach programming without being tied down by the limitations of existing tools and languages?
- Programming is almost always taught as a craft in the context of current technology (e.g., Java and its tools)
  - Any science given is either limited to the current technology or is too theoretical
- The concepts-based approach shows one way to solve this problem
How can we teach programming paradigms?

- Different languages support different paradigms
  - **Java**: object-oriented programming
  - **Haskell**: functional programming
  - **Erlang**: concurrent programming (for reliability)
  - **Prolog**: logic programming
  - ...
- We would like to understand all these paradigms!
  - They are all important and practical
- Does this mean we have to study as many languages?
  - New syntaxes to learn …
  - New semantics to learn …
  - New systems to learn …
- No!
Our pragmatic solution

- Use the concepts-based approach
  - With Oz as the single language
  - With Mozart as the single system
- This supports all the paradigms we want to teach
  - But we are not dogmatic about Oz
  - We use it because it fits the approach well
- We situate other languages inside our general framework
  - We can give a deep understanding rather quickly, for example:
    - Visibility rules of Java and C++
    - Inner classes of Java
    - Good programming style in Prolog
    - Message receiving in Erlang
    - Lazy programming style in Haskell
Teaching with the concepts-based approach (1)

- We show languages in a progressive way
  - We start with a small language containing just a few programming concepts
  - We show how to program and reason in this language
  - We then add concepts one by one to remove limitations in expressiveness

- In this way we cover all major programming paradigms
  - We show how they are related and how and when to use them together
Teaching with the concepts-based approach (2)

- Similar approaches have been used before
  - Notably by Abelson & Sussman in “Structure and Interpretation of Computer Programs”
- We apply the approach both broader and deeper: we cover more paradigms and we have a simple formal semantics for all concepts
- We have especially good coverage of concurrency and data abstraction
Some courses (1)

- Second-year course (Datalogi II at KTH, CS2104 at NUS) by Seif Haridi and Christian Schulte
  - Start with **declarative programming**
  - Explain declarative techniques and higher-order programming
  - Explain semantics
  - Add **threads**: leads to declarative concurrency
  - Add **ports** (communication channels): leads to message-passing concurrency (agents)
- **Declarative programming, concurrency, and multi-agent systems**
  - For deep reasons, this is a better start than OOP
Some courses (2)

- Second-year course (FSAC1450 at UCL) by Peter Van Roy
  - Start with declarative programming
  - Explain declarative techniques
  - Explain semantics
  - Add cells (mutable state)
  - Explain data abstraction: objects and ADTs
  - Explain object-oriented programming: classes, polymorphism, and inheritance
  - Add threads: leads to declarative concurrency

- Most comprehensive overview in one course
Some courses (3)

- Third-year course (INGI2131 at UCL) by Peter Van Roy
  - Review of declarative programming
  - Add threads: leads to declarative concurrency
    - Add by-need synchronization: leads to lazy execution
    - Combining lazy execution and concurrency
  - Add ports (communication channels): leads to message-passing concurrency
    - Designing multi-agent systems
  - Add cells (mutable state): leads to shared-state concurrency
    - Tuple spaces (Linda-like)
    - Locks, monitors, transactions
- Concurrency in all its manifestations
Examples showing the usefulness of the approach

- The concepts-based approach gives a broader and deeper view of programming than the more traditional language- or tool-oriented approach.
- Let us see some examples of this:
  - Concurrent programming
  - Data abstraction
  - Graphical user interface programming
  - Object-oriented programming in a wider framework
- We explain these examples
Concurrent programming

- There are three main paradigms of concurrent programming
  - **Declarative (dataflow; deterministic) concurrency**
  - **Message-passing concurrency** (active entities that send asynchronous messages; Erlang style)
  - **Shared-state concurrency** (active entities that share common data using locks and monitors; Java style)
- **Declarative concurrency** is very useful, yet is little known
  - No race conditions; declarative reasoning techniques
  - Large parts of programs can be written with it
- **Shared-state concurrency** is the most complicated, yet it is the most widespread!
  - Message-passing concurrency is a better default
Example of declarative concurrency

- Producer/consumer with dataflow

fun {Prod N Max}
  if N<Max then
  N|{Prod N+1 Max}
  else nil end
end

proc {Cons Xs}
  case Xs of X|Xr then
    {Display X}
    {Cons Xr}
    [] nil then skip end
end

local Xs in
  thread Xs={Prod 0 1000} end
  thread {Cons Xs} end
end

- Prod and Cons threads share dataflow
- List Xs
- Dataflow behavior of case statement (synchronize on data availability) gives stream communication
- No other concurrency control needed
Data abstraction

• A data abstraction is a high-level view of data
  • It consists of a set of instances, called the data, that can be manipulated according to certain rules, called the interface
  • The advantages of this are well-known, e.g., it is simpler to use, it segregates responsibilities, it simplifies maintenance, and the implementation can provide some behavior guarantees

• There are at least four ways to organize a data abstraction
  • According to two axes: bundling and state
Objects and ADTs

- The first axis is bundling
- An abstract data type (ADT) has separate values and operations
  - Example: integers (values: 1, 2, 3, …; operations: +, -, *, div, …)
  - Canonical language: CLU (Barbara Liskov et al, 1970s)
- An object combines values and operations into a single entity
  - Example: stack objects (instances with push, pop, isEmpty operations)
  - Canonical language: Smalltalk (Xerox PARC, 1970s)
Have objects won?

- Absolutely not! Currently popular “object-oriented” languages actually mix objects and ADTs
  - For example, in Java:
    - Basic types such as integers are ADTs (which is nothing to apologize about)
    - Instances of the same class can access each other’s private attributes (which is an ADT property)
  - To understand these languages, it’s important for students to understand objects and ADTs
    - ADTs allow to express efficient implementation, which is not possible with pure objects (even Smalltalk is based on ADTs!)
    - Polymorphism and inheritance work for both objects and ADTs, but are easier to express with objects
- For more information and explanation, see the book!
Summary of data abstractions

- The book explains how to program these four possibilities and says what they are good for.

The usual one!
Graphical user interface programming

- There are three main approaches:
  - Imperative approach (AWT, Swing, tcl/tk, ...): maximum expressiveness with maximum development cost
  - Declarative approach (HTML): reduced development cost with reduced expressiveness
  - Interface builder approach: adequate for the part of the GUI that is known before the application runs

- All are unsatisfactory for dynamic GUIs, which change during execution
Mixed declarative/imperative approach to GUI design

- Using both approaches together is a plus:
  - A declarative specification is a data structure. It is concise and can be calculated in the language.
  - An imperative specification is a program. It has maximum expressiveness but is hard to manipulate formally.

- This makes creating dynamic GUls very easy
- This is an important foundation for model-based GUI design, an important methodology for human-computer interfaces
Example GUI

Nested record with handler object \( E \) and action procedure \( P \)

Construct interface (window & handler object)

Call the handler object

\[
W = \text{td}((l(r(\text{label}(\text{text:"Enter your name"}) \\
\text{entry}(\text{handle}:E)) \\
\text{button}(\text{text:"Ok" action}:P)))
\]

\[
\{\text{Build } W\}
\]

\[
\{E \text{ set}(\text{text:"Type here"})\}
\]

\[
\text{Result} = \{E \text{ get}(\text{text:$})\}
\]
Example dynamic GUI

\[W = \text{placeholder}(\text{handle;} P)\]

\[\ldots\]

\[\{P \text{ set( } \text{label(text:"Hello") }) \}\]

\[\{P \text{ set( } \text{entry(text:"World") }) \}\]

- Any GUI specification can be put in the placeholder at run-time (the spec is a data structure that can be calculated)
Object-oriented programming: a small part of a big world

- Object-oriented programming is just one tool in a vastly bigger world
- For example, consider the task of building robust telecommunications systems
  - Ericsson has developed a highly available ATM switch, the AXD 301, using a message-passing architecture (more than one million lines of Erlang code)
  - The important concepts are isolation, concurrency, and higher-order programming
  - Not used are inheritance, classes and methods, UML diagrams, and monitors
Formal semantics

- It’s important to put programming on a solid foundation. Otherwise students will have muddled thinking for the rest of their careers.
  - Typical mistake: confusing syntax and semantics
- We propose a flexible approach, where more or less semantics can be given depending on your taste and the course goals
  - The foundation of all the different semantics is an operational semantics, an abstract machine
Three levels of teaching semantics

- **First level**: abstract machine (the rest of this talk)
  - Concepts of execution stack and environment
  - Can explain last call optimization and memory management (including garbage collection)

- **Second level**: structural operational semantics
  - Straightforward way to give semantics of a practical language
  - Directly related to the abstract machine

- **Third level**: develop the mathematical theory
  - Axiomatic, denotational, and logical semantics are introduced for the paradigms in which they work best
  - Primarily for theoretical computer scientists
Abstract machine

- The approach has three steps:
  - Full language: includes all syntactic support to help the programmer
  - Kernel language: contains all the concepts but no syntactic support
  - Abstract machine: execution of programs written in the kernel language
fun \{Fact N\}
    if N==0 then 1
else N*\{Fact N-1\}
end

proc \{Fact N F\}
    local B in
        B=(N==0)
        if B then F=1
else
    local N1 F1 in
        N1=N-1
        \{Fact N1 F1\}
        F=N*F1
    end
end
end

All syntactic aids are removed: all identifiers are shown (locals and output arguments), all functions become procedures, etc.
Syntax of a simple kernel language (1)

- EBNF notation; \(<s>\) denotes a statement

\[
\begin{align*}
<s> & ::= \text{skip} \\
 & \quad | \text{\(x\)}_1 = \text{\(x\)}_2 \\
 & \quad | \text{\(x\)} = \text{\(v\)} \\
 & \quad | \text{local} \text{\(x\)} \text{ in } <s> \text{ end} \\
 & \quad | \text{if} \text{\(x\)} \text{ then } <s>_1 \text{ else } <s>_2 \text{ end} \\
 & \quad | \{\text{\(x\)} \text{ \(x\)}_1 \ldots \text{\(x\)}_n\} \\
 & \quad | \text{case} \text{\(x\)} \text{ of } \text{\(p\)} \text{ then } <s>_1 \text{ else } <s>_2 \text{ end}
\end{align*}
\]

\[
\begin{align*}
\text{\(v\)} & ::= \ldots \\
\text{\(p\)} & ::= \ldots
\end{align*}
\]
Syntax of a simple kernel language (2)

- EBNF notation; \(<v>\) denotes a value, \(<p>\) denotes a pattern

\[
\begin{align*}
<v> & ::= <record> | <number> | <procedure> \\
<record>, <p> & ::= <lit> | <lit>(<feat>_{1}::<x>_{1} \ldots <feat>_{n}::<x>_{n}) \\
<number> & ::= <int> | <float> \\
<precedure> & ::= \textproc\{<x>_{1} \ldots <x>_{n}\} <s> \text{end}
\end{align*}
\]

- This kernel language covers a simple declarative paradigm
- Note that it is definitely not a “theoretically minimal” language!
  - It is designed to be simple for programmers, not to be mathematically minimal
  - This is an important principle throughout the book!
  - We want to show programming techniques
  - But the semantics is still simple and usable for reasoning
Abstract machine concepts

- Single-assignment store $\sigma = \{x_1 = 10, x_2, x_3 = 20\}$
  - Variables and their values
- Environment $E = \{X \rightarrow x, Y \rightarrow y\}$
  - Link between program identifiers and store variables
- Semantic statement $(<s>, E)$
  - A statement with its environment
- Semantic stack $ST = [(<s_1>, E_1), \ldots, (<s_n>, E_n)]$
  - A stack of semantic statements, “what remains to be done”
- Execution $(ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow (ST_3, \sigma_3) \rightarrow \ldots$
  - A sequence of execution states (stack + store)
The local statement

- \((\text{local } X \text{ in } \langle s \rangle \text{ end}, E)\)
  - Create a new store variable \(x\)
  - Add the mapping \(\{X \rightarrow x\}\) to the environment

<table>
<thead>
<tr>
<th>Stack</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S_2)</td>
<td>(\sigma)</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\sigma \cup {x})</td>
</tr>
<tr>
<td>(S_n)</td>
<td>(\sigma \cup {x})</td>
</tr>
</tbody>
</table>
The if statement

- (if <x> then <s>_1 else <s>_2 end, E)
- This statement has an activation condition: \( E(<x>) \) must be bound to a value
- Execution consists of the following actions:
  - If the activation condition is true, then do:
    - If \( E(<x>) \) is not a boolean, 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 the execution does nothing (it suspends)
- If some other activity makes the activation condition true, then execution continues. This gives dataflow synchronization, which is at the heart of declarative concurrency.
Procedures (closures)

• A procedure value (closure) is a pair
  \[(\text{proc} \ \{\$ \ <y>_1 \ldots \ <y>_n\} \ <s> \ \text{end}, \ CE)\]
  where \(CE\) (the “contextual environment”) is \(E|_{<z>_1,\ldots,<z>_n}\) with \(E\) the environment where the procedure is defined and \({<z>_1, \ldots, <z>_n}\) the set of the procedure’s external identifiers

• A procedure call \(\{<x> <x>_1 \ldots <x>_n\}, E\) executes as follows:
  - If \(E(<x>)\) is a procedure value as above, then push
    \((<s>, CE+\{<y>_1\rightarrow E(<x>_1), \ldots, <y>_n\rightarrow E(<x>_n)\})\)
    on the semantic stack

• This allows higher-order programming as in functional languages
Use of the abstract machine

- With it, students can work through program execution at the right level of detail
  - Detailed enough to explain many important properties
  - Abstract enough to make it practical and machine-independent (e.g., we do not go down to the machine architecture level!)
- We use it to explain behavior and derive properties
  - We explain last call optimization
  - We explain garbage collection
  - We calculate time and space complexity of programs
  - We explain higher-order programming
  - We give a simple semantics for objects and inheritance
Conclusions

- We presented the **concepts-based approach**, one way to organize the discipline of computer programming
  - Programming languages are organized according to their concepts
  - New concepts are added to overcome limitations in expressiveness *(creative extension principle)*
  - The complete set of concepts covers all major programming paradigms
- We gave examples of how this approach **gives insight**
  - Concurrent programming, data abstraction, GUI programming, the role of object-oriented programming
- We have written a **textbook** based on this approach and are using it to teach second-year to graduate courses
  - The textbook covers both theory (formal semantics) and practice (using the Mozart Programming System)
  - The textbook is based on research done in the Mozart Consortium
  - See also Second Int’l Mozart/Oz Conference (Springer LNAI 3389)