A Concepts-Based Approach for Teaching Programming

November 2005

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

- Teaching programming
  - What is programming?
  - Concepts-based approach
  - Courses and a textbook
- Foundations of the concepts-based approach
  - History
  - Creative extension principle
- Examples of the concepts-based approach
  - Concurrent programming
  - Data abstraction
  - Graphical user interface programming
  - Object-oriented programming: a small part of a big world
- Teaching formal semantics
- Conclusion
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)
  - The science taught is either limited to currently popular technology or is too theoretical
- We would like to teach programming in a more scientific way: as a unified discipline that is both practical and theoretically sound
- The concepts-based approach shows one way to achieve this
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
  - The major programming paradigms appear one by one. We show how they are related and how and when to use them together.

- We have applied this approach in a much broader and deeper way than has been done before (e.g., by Abelson & Sussman)
  - Based on research results from a long-term collaboration
Teaching programming paradigms

- Different languages support different paradigms
  - Java, C#, Python, Ruby: object-oriented programming
  - Haskell, ML, Scheme: functional programming
  - Erlang: concurrent and distributed programming (for reliability)
  - Prolog, Mercury: logic programming
  - Many more languages and paradigms are used in industry

- We would like to understand these languages and paradigms
  - They are all important and practical

- Does this mean we have to study each of them separately?
  - New syntaxes to learn …
  - New semantics to learn …
  - New systems to install and learn …

- No!
Our pragmatic solution

- Use the concepts-based approach
  - With Oz as single language
  - With the Mozart Programming System as 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 techniques in Haskell
The textbook

- We have written a textbook to support the approach
  - The textbook is based on more than a decade of research by an international group, the Mozart Consortium

- Goals of the textbook
  - 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
Some courses (1)

- Second-year course (Datalog I 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 (dataflow) concurrency
  - Add ports (communication channels): leads to message-passing concurrency (agents)
- Declarative programming, concurrency, and multi-agent systems
  - For many reasons, this is a better start than OOP
Some courses (2)

- Second-year course (FSAC1450, LINF1251, FSAB1402 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 and stream objects

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

- Third-year course (INGI2131 at UCL) by Peter Van Roy
  - Review of declarative programming
  - Add **threads**: leads to declarative concurrency and stream objects
    - 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
- **Focus on concurrent programming**
A more advanced course (4)

- This is an example of a graduate course given at an American university
- It covers many more paradigms, their semantics, and some of their relationships
Curriculum insight: second year is best!

- In our experience, the best place to put a broad course on programming concepts is in the second year
  - The first year course is a basic introduction to programming; it can be based on almost any modern language, such as Java
  - In the third and fourth years, students become very conservative. If they have been programming in Java up to then, they will resist learning new concepts.
- In the second year, students already know what programming is, but their minds are still open
  - They readily accept a broad course on programming concepts
  - This is also true for non computer science majors
- We base this conclusion on four years of experience with the concepts-based approach, teaching students in the second, third, and fourth years in several major universities (UCL, NUS, KTH)
Foundations of the concepts-based approach
History: the ancestry of Oz

- 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 Japanese 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 language 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 Programming System**
History: teaching with Oz

- In the summer of 1999, we (Seif Haridi and Peter Van Roy) had an epiphany: we realized that we 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. It appeared in 2004 from MIT Press.
- 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 much important computer science that was forgotten, e.g., deterministic concurrency, objects vs. ADTs
  - Both were already known in the 1970s, but largely ignored afterward!
Creative extension principle

- Design language by overcoming 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

\[
\text{proc } \{P1 \ldots E1\} \\
\{P2 \ldots E2\} \\
\text{if } E2 \text{ then } \ldots \text{ end} \\
E1=\
\text{end} \\
\text{proc } \{P2 \ldots E2\} \\
\{P3 \ldots E3\} \\
\text{if } E3 \text{ then } \ldots \text{ end} \\
E2=\
\text{end} \\
\text{proc } \{P3 \ldots E3\} \\
\{P4 \ldots E4\} \\
\text{if } E4 \text{ then } \ldots \text{ end} \\
E3=\
\text{end} \\
\text{proc } \{P4 \ldots E4\} \\
\text{if (error) then } E4=\text{true} \\
\text{else } E4=\text{false} \text{ end} \\
\text{end}
\]

Language with exceptions

\[
\text{proc } \{P1 \ldots\} \\
\text{try} \\
\{P2 \ldots\} \\
\text{catch } E \text{ then } \ldots \text{ end} \\
\text{end} \\
\text{end} \\
\text{proc } \{P2 \ldots\} \\
\{P3 \ldots\} \\
\text{end} \\
\text{proc } \{P3 \ldots\} \\
\{P4 \ldots\} \\
\text{end}
\]

All procedures on path are modified

Error treated here

Error occurs here

Only procedures at ends are modified

Error treated here

Error occurs here

Unchanged

Error occurs here

Unchanged
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
- Concurrent OOP
  (message passing, Erlang, E)
  (shared state, Java, C#)
- Object-oriented programming, Java, C++, C#
  + search
- Nondeterministic logic prog., Prolog
  + computation spaces
- Constraint programming

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

\[
\begin{align*}
<s> & ::= \\
& \quad \text{skip} \\
& \quad <x>_1 = <x>_2 \\
& \quad <x> = \text{record} \mid \text{number} \mid \text{procedure} \\
& \quad <s>_1 \ <s>_2 \\
& \quad \text{local} <x> \ \text{in} \ <s> \ \text{end} \\
& \quad \text{if} <x> \ \text{then} <s>_1 \ \text{else} <s>_2 \ \text{end} \\
& \quad \text{case} <x> \ \text{of} \ <p> \ \text{then} <s>_1 \ \text{else} <s>_2 \ \text{end} \\
& \quad \{<x> \ <x>_1 \ \ldots \ <x>_n\} \\
& \quad \text{thread} <s> \ \text{end} \\
& \quad \{\text{WaitNeeded} <x>\} \\
& \quad \{\text{Name creation} <x>\} \\
& \quad <x>_1 = !!<x>_2 \\
& \quad \text{try} <s>_1 \ \text{catch} <x> \ \text{then} <s>_2 \ \text{end} \\
& \quad \text{raise} <x> \ \text{end} \\
& \quad \{\text{NewCell} <x>_1 <x>_2\} \\
& \quad \{\text{Exchange} <x>_1 <x>_2 <x>_3\} \\
& \quad <\text{space}> \\
& \quad \text{Encapsulated search}
\end{align*}
\]
Examples of the concepts-based approach
Examples showing the usefulness of the approach

- The concepts-based approach gives a broader and deeper view of programming than more traditional language- or tool-oriented approaches.
- We illustrate this with four examples:
  - Concurrent programming
  - Data abstraction
  - Graphical user interface programming
  - Object-oriented programming in a wider framework
Concurrent programming

- There are three main paradigms of concurrent programming
  - Declarative (dataflow; deterministic) concurrency
  - Message-passing concurrency (active entities that send asynchronous messages; Actor model, 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; allows declarative reasoning techniques
  - Large parts of programs can be written with it
- Shared-state concurrency is the most complicated (the worst to program in), yet it is the most widespread!
  - Message-passing concurrency is a better default
Example of declarative concurrency

- Producer/consumer with dataflow

```haskell
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 stream `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 and learn, it segregates responsibilities (team projects), 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 the operations push, pop, isEmpty)
  - Canonical languages: Simula (Dahl & Nygaard, 1960s), Smalltalk (Xerox PARC, 1970s), C++ (Stroustrup, 1980)
Summary of data abstractions

- The book explains how to program these four possibilities and says what they are good for.
Have objects defeated ADTs?

- Absolutely not! Currently popular “object-oriented” languages actually mix objects and ADTs, for good reasons
  - For example, in Java:
    - Basic types such as integers are ADTs (a perfectly good design decision)
    - 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!
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
  - For example, display characteristics often change during and between executions
Mixed declarative/imperative approach to GUI design

- Using both approaches together can get the best of both worlds:
  - A declarative specification is a data structure. It is concise and can be calculated by a program.
  - An imperative specification is a program. It has maximum expressiveness but is hard to manipulate as a data structure.
- This makes creating dynamic GUIs very easy
- This is an important foundation for model-based GUI design, an important methodology for human-computer interfaces
Example GUI

```
W = td(lr(label(text:"Enter your name"),
entry(handle:E),
button(text:"Ok" action:P))

{Build W}

{E set(text:"Type here")}
Result={E get(text:$.})
```
Example dynamic GUI

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

... 

\{ P \text{ set}( \text{label}(\text{text:”Hello”}) ) \}\}

\{ P \text{ set}( \text{entry}(\text{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
Teaching formal semantics
Teaching formal semantics

- It’s important to put programming on a solid foundation. Otherwise students will have muddled thinking for the rest of their careers.
  - A common mistake is confusing syntax and semantics
  - A simple semantics is important for predictable and intuitive behavior, even if the students don’t learn it
- But how can we teach semantics without getting bogged down by the mathematics?
  - The semantics should be simple enough to be used by programmers, not just by mathematicians
- We propose an approach based on a simple abstract machine (an operational semantics)
  - It is both simple and rigorous
  - We teach it successfully to second-year engineering students
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 *(CTM chapter 13)*

- 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
First level: 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
Translating to kernel language

fun \{\text{Fact } N\} 
  \text{if } N == 0 \text{ then } 1 
  \text{else } N \times \{\text{Fact } N-1\} 
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

```plaintext
<s> ::= skip
| <x>₁=<x>₂
| <x>==<v>
| local <x> in <s> end
| if <x> then <s>₁ else <s>₂ end
| {<x> <x>₁ ... <x>ₙ}
| case <x> of <p> then <s>₁ else <s>₂ end
```

```plaintext
<v> ::= ...
<p> ::= ...
```
Syntax of a simple kernel language (2)

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

\[
\begin{align*}
\langle v \rangle &::= \langle \text{record} \rangle \mid \langle \text{number} \rangle \mid \langle \text{procedure} \rangle \\
\langle \text{record} \rangle, \langle p \rangle &::= \langle \text{lit} \rangle \mid \langle \text{lit} \rangle(\langle \text{feat} \rangle_1:x_1 \ldots \langle \text{feat} \rangle_n:x_n) \\
\langle \text{number} \rangle &::= \langle \text{int} \rangle \mid \langle \text{float} \rangle \\
\langle \text{procedure} \rangle &::= \text{proc} \{\$ \langle x \rangle_1 \ldots \langle x \rangle_n\} <s> \text{ end}
\end{align*}
\]

- This kernel language covers a simple declarative paradigm
- Note that it is definitely not a “theoretically minimal” language!
  - It is designed for programmers, not for mathematicians
  - This is an important principle throughout the book!
  - We want to teach programming, not mathematics
  - The semantics is both intuitive and useful for reasoning
Abstract machine concepts

- Single-assignment store $\sigma = \{x_1 = 10, x_2, x_3 = 20\}$
  - Memory variables and their values
- Environment $E = \{X \rightarrow x, Y \rightarrow y\}$
  - Link between program identifiers and store variables
- Semantic statement $\langle s \rangle, E$
  - A statement with its environment
- Semantic stack $ST = [\langle s \rangle_1, E_1, \ldots, \langle s \rangle_n, E_n]$,
  - 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 memory variable \(x\)
  - Add the mapping \(\{X \rightarrow x\}\) to the environment

<table>
<thead>
<tr>
<th>(\text{(local } X \text{ in } \langle s \rangle \text{ end}, E))</th>
<th>(\sigma)</th>
<th>(\sigma \cup {x})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S_2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(S_n)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(\langle s \rangle, E+{X \rightarrow x})</th>
<th>(\sigma)</th>
<th>(\sigma \cup {x})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S_2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(S_n)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The if statement

- \((\text{if } <x> \text{ then } <s>_1 \text{ else } <s>_2 \text{ end, } E)\)
- 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
  \[(\text{proc} \{\text{s} <y>_1 \ldots <y>_n\} \text{s} \text{ end}, CE)\]
  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 procedure’s free 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.
Using 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 a *concepts-based approach* for teaching programming
  - Programming languages are organized according to their concepts
  - The full set of concepts covers all major programming paradigms
- We gave examples of how this approach *gives insight*
  - Concurrent programming, data abstraction, GUI programming, the true role of object-oriented programming
- We have written a *textbook* published by MIT Press in 2004 and are using it to teach second-year to graduate courses
  - “Concepts, Techniques, and Models of Computer Programming”
  - The textbook covers both theory (formal semantics) and practice (using the Mozart Programming System, [http://www.mozart-oz.org](http://www.mozart-oz.org))
- For more information
  - Also *Multiparadigm Programming in Mozart/Oz* (Springer LNCS 3389)