A Concepts-Based Approach for Teaching Programming

SIGCSE 2005 Birds of a Feather Session

Feb. 24, 2005

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

pvr@info.ucl.ac.be
http://www.info.ucl.ac.be/people/PVR/book.html
http://www.mozart-oz.org

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)
  - Any science given is either limited to the current technology or is too theoretical
- We would like to teach programming 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
  - We show how the major programming paradigms 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

How can we teach programming paradigms?

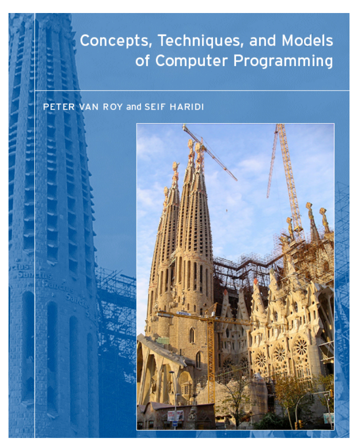
- Different languages support different paradigms
  - Java: object-oriented programming
  - Haskell: functional programming
  - Erlang: concurrent and distributed programming (for reliability)
  - Prolog: logic programming
  - Many more languages and paradigms are used in industry
- We would like to 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 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

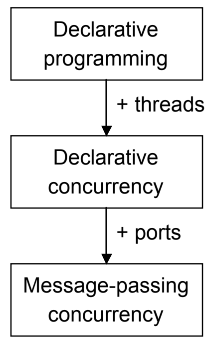
The textbook

- We have written a textbook to support the approach
  - "Concepts, Techniques, and Models of Computer Programming", MIT Press, 2004
  - 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



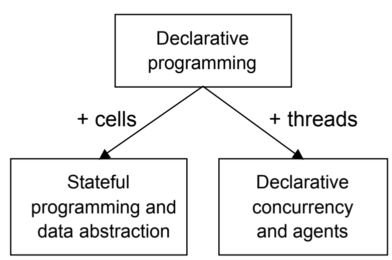
Some courses (1)

- Second-year course (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 (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



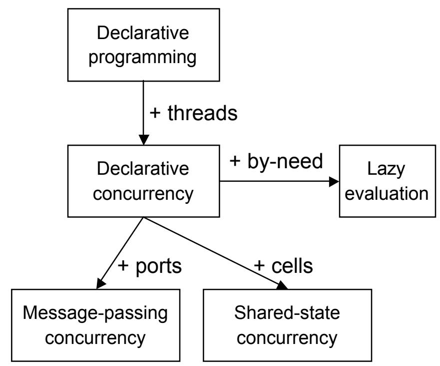
Some courses (2)

- Second-year course (FSAC1450 and LINF1251 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 early course

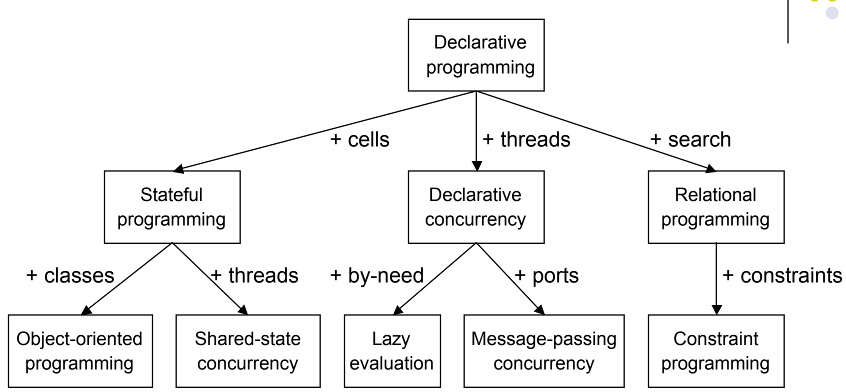


Some courses (3)

- Third-year course (INGI2131 at UCL) by Peter Van Roy
  - Review of declarative programming
  - Add threads: leads to declarative concurrency
    - 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



- This is an example of a more advanced course given at an unnamed institution
- It covers many more paradigms, their semantics, and some of their relationships

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 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

History: teaching with Oz

- In the summer of 1999, we (the authors) 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

- 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
end
```

Error treated here

All procedures on path are modified

Error occurs here

Language with exceptions

```
proc {P1 …}
  try
    {P2 …}
  catch E then … end
end

proc {P2 …}
  {P3 …}
end

proc {P3 …}
  {P4 …}
end

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

Error treated here

Only procedures at ends are modified

Unchanged

Error occurs here

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

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

+ 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> | <number> | <procedure>	*Value creation*
<s>$_1$ <s>$_2$	*Sequential composition*
local <x> **in** <s> **end**	*Variable creation*
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)

<s> ::=

skip	*Empty statement*
<x>$_1$=<x>$_2$	*Variable binding*
<x>=<record> | <number> | <procedure>	*Value creation*
<s>$_1$ <s>$_2$	*Sequential composition*
local <x> **in** <s> **end**	*Variable creation*
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*
{NewCell <x>$_1$ <x>$_2$}	*Cell creation* } **Alternative**
{Exchange <x>$_1$ <x>$_2$ <x>$_3$}	*Cell exchange*
<space>	*Encapsulated search*

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

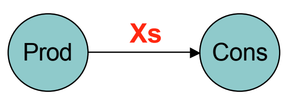
Concurrent programming

- 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 and 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 stream **Xs**
- Dataflow behavior of case statement (synchronize on data availability) gives stream communication
- No other concurrency control needed

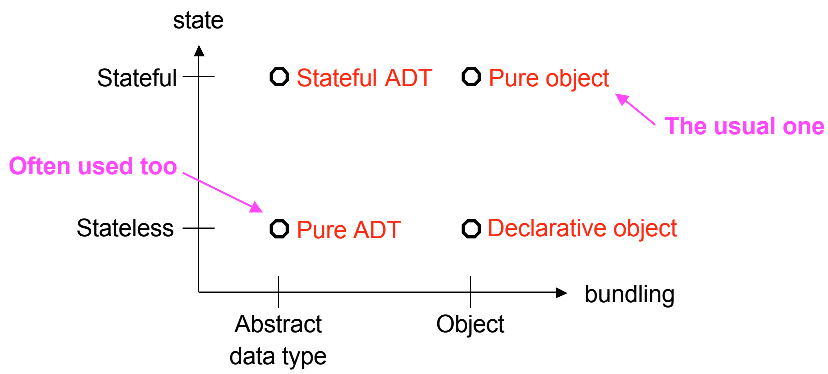
Data abstraction

- 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 push, pop, isEmpty operations)
  - Canonical languages: Simula (Dahl & Nygaard, 1960s), Smalltalk (Xerox PARC, 1970s)

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

Nested record with handler object E and action procedure P → W=td(lr(label(text:"Enter your name")
entry(handle:E))
button(text:"Ok" action:P))

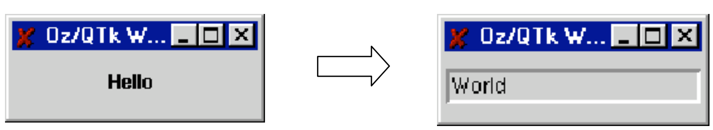
…

Construct interface (window & handler object) → {Build W}

…

Call the handler object → {E set(text:"Type here")}
Result={E get(text:$)}

Example dynamic GUI



W=placeholder(handle:P)

…

{P set(**label(text:"Hello")**)}

{P set(**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

Teaching formal semantics

Teaching formal semantics

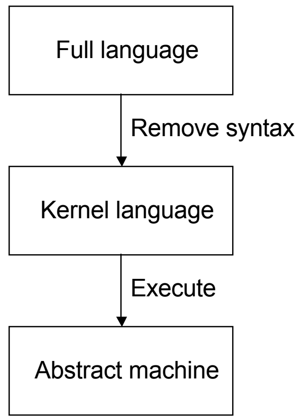
- It's important to put programming on a solid foundation. Otherwise students will have muddled thinking for the rest of their careers.
  - A typical mistake is confusing syntax and semantics
  - A simple semantics is important for predictable and intuitive behavior, even if the students don't learn 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
- 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



Translating to kernel language

```
fun {Fact N}
   if N==0 then 1
   else N*{Fact N-1}
   end
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
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

<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**

<v> ::= …
<p> ::= …

Syntax of a simple kernel language (2)

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

<v> ::= <record> | <number> | <procedure>
<record>, <p> ::= <lit> | <lit>(<feat>$_1$:<x>$_1$ … <feat>$_n$:<x>$_n$)
<number> ::= <int> | <float>
<procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**

- 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

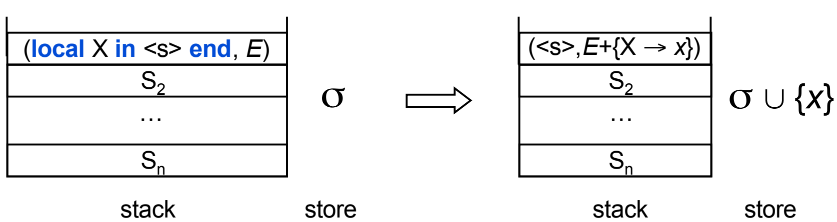
Abstract machine concepts

- 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 (<s>,E)
  - A statement with its environment
- Semantic stack ST = [(<s>$_1$,E$_1$), …, (<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)

Feb. 24, 2005 P. Van Roy, SIGCSE 2005, BoF session 41

The **local** statement

- (**local** X **in** <s> **end**, *E*)
  - Create a new memory variable *x*
  - Add the mapping {X → *x*} to the environment



Feb. 24, 2005 P. Van Roy, SIGCSE 2005, BoF session 42

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
  (**proc** {$ <y>$_1$ … <y>$_n$} <s> **end**, *CE*)
  where *CE* (the "contextual environment") is $E|_{\langle z\rangle 1, \ldots, \langle z\rangle n}$ with
  E the environment where the procedure is defined and
  {<z>$_1$, …, <z>$_n$} the procedure's free identifiers
- A procedure call ({<x> <x>$_1$ … <x>$_n$}, *E*) executes as follows:
  - If *E*(<x>) is a procedure value as above, then push
    (<s>, *CE*+{<y>$_1$→*E*(<x>$_1$), …, <y>$_n$→*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 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)
- For more information
  - See http://www.info.ucl.ac.be/people/PVR/book.html
  - Also Multiparadigm Programming in Mozart/Oz (Springer LNCS 3389)