Introduction to Programming Concepts (VRH 1.9-1.17)

Carlos Varela
RPI
February 25, 2010

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL
Introduction

• An introduction to programming concepts
• Declarative variables
• Functions
• Structured data (example: lists)
• Functions over lists
• Correctness and complexity
• Lazy functions
• Higher-order programming
• Concurrency and dataflow
• State, objects, and classes
• Nondeterminism and atomicity
Higher-order programming

• Assume we want to write another Pascal function, which instead of adding numbers, performs exclusive-or on them
• It calculates for each number whether it is odd or even (parity)
• Either write a new function each time we need a new operation, or write one generic function that takes an operation (another function) as argument
• The ability to pass functions as arguments, or return a function as a result is called higher-order programming
• Higher-order programming is an aid to build generic abstractions
Variations of Pascal

• Compute the parity Pascal triangle

fun {Xor X Y} if X==Y then 0 else 1 end end

\[
\begin{array}{cccccc}
1 & & & & & 1 \\
1 & 1 & & & & 1 \\
1 & 2 & 1 & & & 1 \\
1 & 3 & 3 & 1 & & 1 \\
1 & 4 & 6 & 4 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]
Higher-order programming

<table>
<thead>
<tr>
<th>fun {GenericPascal Op N}</th>
<th>fun {Add N1 N2} N1+N2 end</th>
</tr>
</thead>
<tbody>
<tr>
<td>if N==1 then [1]</td>
<td></td>
</tr>
<tr>
<td>else L in L = {GenericPascal Op N-1}</td>
<td></td>
</tr>
<tr>
<td>{OpList Op {ShiftLeft L} {ShiftRight L}}</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>fun {OpList Op L1 L2}</td>
<td>fun {Xor N1 N2}</td>
</tr>
<tr>
<td>case L1 of H1</td>
<td>T1 then</td>
</tr>
<tr>
<td>case L2 of H2</td>
<td>T2 then</td>
</tr>
<tr>
<td>{Op H1 H2}</td>
<td>{OpList Op T1 T2}</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>else nil end</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

fun {Pascal N} {GenericPascal Add N} end

fun {ParityPascal N}

{GenericPascal Xor N}
Concurrency

- How to do several things at once
- Concurrency: running several activities each running at its own pace
- A *thread* is an executing sequential program
- A program can have multiple threads by using the thread instruction
- {Browse 99*99} can immediately respond while Pascal is computing

```plaintext
thread
    P in
    P = {Pascal 21}
    {Browse P}
end
{Browse 99*99}
```
Dataflow

- What happens when multiple threads try to communicate?
- A simple way is to make communicating threads synchronize on the availability of data (data-driven execution)
- If an operation tries to use a variable that is not yet bound it will wait
- The variable is called a *dataflow variable*
Dataflow (II)

• Two important properties of dataflow
  – Calculations work correctly independent of how they are partitioned between threads (concurrent activities)
  – Calculations are patient, they do not signal error; they wait for data availability

• The dataflow property of variables makes sense when programs are composed of multiple threads

```
declare X
thread
  {Delay 5000} X=99
End
{Browse ‘Start’} {Browse X*X}
```

```
declare X
thread
  {Browse ‘Start’} {Browse X*X}
end
{Delay 5000} X=99
```
State

- How to make a function learn from its past?
- We would like to add memory to a function to remember past results
- Adding memory as well as concurrency is an essential aspect of modeling the real world
- Consider \{FastPascal N\}: we would like it to remember the previous rows it calculated in order to avoid recalculating them
- We need a concept (memory cell) to store, change and retrieve a value
- The simplest concept is a (memory) cell which is a container of a value
- One can create a cell, assign a value to a cell, and access the current value of the cell
- Cells are not variables

declare
C = \{NewCell 0\}
\{Assign C \{Access C\}+1\}
\{Browse \{Access C\}\}
Example

- Add memory to Pascal to remember how many times it is called
- The memory (state) is global here
- Memory that is local to a function is called *encapsulated state*

```pascal
declare
C = {NewCell 0}
fun {FastPascal N}
    {Assign C {Access C}+1}
    {GenericPascal Add N}
end
```
Objects

- Functions with internal memory are called *objects*
- The cell is invisible outside of the definition

```plaintext
declare
local C in
C = {NewCell 0}
fun {Bump}
  {Assign C {Access C}+1}
  {Access C}
end
end
```

```plaintext
declare
fun {FastPascal N}
  {Browse {Bump}}
  {GenericPascal Add N}
end
```
Classes

- A class is a ’factory’ of objects where each object has its own internal state
- Let us create many independent counter objects with the same behavior

```plaintext
fun {NewCounter}
  local C Bump in
  C = {NewCell 0}
  fun {Bump}
    {Assign C {Access C}+1}
    {Access C}
  end
  Bump
end
end
```
Here is a class with two operations: Bump and Read

```plaintext
fun {NewCounter}
  local C Bump Read in
  C = {NewCell 0}
  fun {Bump}
    {Assign C {Access C}+1}
    {Access C}
  end
  fun {Read}
    {Access C}
  end
[bump read]
end
end
```
Object-oriented programming

• In object-oriented programming the idea of objects and classes is pushed farther

• Classes keep the basic properties of:
  – State encapsulation
  – Object factories

• Classes are extended with more sophisticated properties:
  – They have *multiple* operations (called *methods*)
  – They can be defined by taking another class and extending it slightly (*inheritance*)
Nondeterminism

- What happens if a program has both concurrency and state together?
- This is very tricky
- The same program can give different results from one execution to the next
- This variability is called *nondeterminism*
- Internal nondeterminism is not a problem if it is not observable from outside
Nondeterminism (2)

declare
C = {NewCell 0}

thread {Assign C 1} end

thread {Assign C 2} end

t_0
C = {NewCell 0}
cell C contains 0

t_1
{Assign C 1}
cell C contains 1

t_2
{Assign C 2}
cell C contains 2 (final value)
Nondeterminism (3)

declare
C = {NewCell 0}

thread {Assign C 1} end
thread {Assign C 2} end

t₀  C = {NewCell 0}
cell C contains 0

t₁  {Assign C 2}
cell C contains 2

t₂  {Assign C 1}
cell C contains 1 (final value)

time
Nondeterminism (4)

```
declare
C = {NewCell 0}

thread I in
    I = {Access C}
    {Assign C I+1}
end

thread J in
    J = {Access C}
    {Assign C J+1}
end
```

- What are the possible results?
- Both threads increment the cell C by 1
- Expected final result of C is 2
- Is that all?
Another possible final result is the cell C containing the value 1

declare
C = {NewCell 0}

thread I in
    I = {Access C}
    {Assign C I+1}
end

thread J in
    J = {Access C}
    {Assign C J+1}
end

C = {NewCell 0}

I = {Access C}
I equal 0

J = {Access C}
J equal 0

{Assign C J+1}
C contains 1

{Assign C I+1}
C contains 1

C. Varela; Adapted w. permission from S. Haridi and P. Van Roy
Lessons learned

• Combining concurrency and state is tricky
• Complex programs have many possible *interleavings*
• Programming is a question of mastering the interleavings
• Famous bugs in the history of computer technology are due to designers overlooking an interleaving (e.g., the Therac-25 radiation therapy machine giving doses 1000’s of times too high, resulting in death or injury)
• If possible try to avoid concurrency and state together
• Encapsulate state and communicate between threads using dataflow
• Try to master interleavings by using *atomic operations*
Atomicity

• How can we master the interleavings?
• One idea is to reduce the number of interleavings by programming with coarse-grained atomic operations
• An operation is atomic if it is performed as a whole or nothing
• No intermediate (partial) results can be observed by any other concurrent activity
• In simple cases we can use a lock to ensure atomicity of a sequence of operations
• For this we need a new entity (a lock)
Atomicity (2)

\[
\text{declare} \\
L = \{\text{NewLock}\} \\
\text{lock } L \text{ then} \\
\{\text{sequence of ops 1}\} \text{ end} \\
\text{Thread 1} \\
\text{lock } L \text{ then} \\
\{\text{sequence of ops 2}\} \text{ end} \\
\text{Thread 2}
\]
The program

```
declare
C = {NewCell 0}
L = {NewLock}

thread
  lock L then I in
    I = {Access C}
    {Assign C I+1}
  end
end

thread
  lock L then J in
    J = {Access C}
    {Assign C J+1}
  end
end
```

The final result of C is always 2
Memoizing FastPascal

• \{FasterPascal N\} New Version
  1. Make a store S available to FasterPascal
  2. Let K be the number of the rows stored in S (i.e. max row is the K\(^{th}\) row)
  3. if N is less or equal to K retrieve the N\(^{th}\) row from S
  4. Otherwise, compute the rows numbered K+1 to N, and store them in S
  5. Return the N\(^{th}\) row from S

• Viewed from outside (as a black box), this version behaves like the earlier one but faster

```declare
S = \{NewStore\}
\{Put S 2 [1 1]\}
\{Browse \{Get S 2\}\}
\{Browse \{Size S\}\}
```
Exercises

32. VRH Exercise 1.6 (page 24)
   c) Change GenericPascal so that it also receives a number to use as an identity for the operation Op: {GenericPascal Op I N}. For example, you could then use it as:
   {GenericPascal Add 0 N}, or
   {GenericPascal fun {$ X Y} X*Y end 1 N}

33. Prove that the alternative version of Pascal triangle (not using ShiftLeft) is correct. Make AddList and OpList commutative.

34. Write the memoizing Pascal function using the store abstraction (available at store.oz).