Introduction to Programming Concepts (VRH 1.1-1.8)

Carlos Varela
RPI
February 22, 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
• Concurrency and dataflow
• State, objects, and classes
• Nondeterminism and atomicity
Variables

- Variables are short-cuts for values, they cannot be assigned more than once
  
  ```
  declare
  V = 9999*9999
  {Browse V*V}
  ```

- Variable identifiers: is what you type
- Store variable: is part of the memory system
- The `declare` statement creates a store variable and assigns its memory address to the identifier ’V’ in the environment
Functions

• Compute the factorial function:

• Start with the mathematical definition

\[
\text{declare} \\
\text{fun} \{\text{Fact } N\} \\
\quad \text{if } N == 0 \text{ then } 1 \text{ else } N \times \{\text{Fact } N-1\} \text{ end} \\
\text{end}
\]

• Fact is declared in the environment

• Try large factorial \{Browse \{\text{Fact } 100\}\}

\[n! = 1 \times 2 \times \cdots \times (n - 1) \times n\]

\[0! = 1\]

\[n! = n \times (n - 1)! \text{ if } n > 0\]
Composing functions

• Combinations of r items taken from n.
• The number of subsets of size r taken from a set of size n

\[
\binom{n}{r} = \frac{n!}{r!(n-r)!}
\]

```declare
fun {Comb N R}
  {Fact N} div ({Fact R}*{Fact N-R})
end
```

• Example of functional abstraction
Structured data (lists)

- Calculate Pascal triangle
- Write a function that calculates the nth row as one structured value
- A list is a sequence of elements:
  
  \[ [1 \ 4 \ 6 \ 4 \ 1] \]
- The empty list is written nil
- Lists are created by means of "|" (cons)

```declare
H=1
T = [2 3 4 5]
{Browse H|T} % This will show [1 2 3 4 5]
```
• Taking lists apart (selecting components)
• A cons has two components: a head, and a tail

```
declare L = [5 6 7 8]
L.1 gives 5
L.2 give [6 7 8]
```

![Diagram of list structure]
Pattern matching

• Another way to take a list apart is by use of pattern matching with a case instruction

    case L of H|T then {Browse H} {Browse T}
      else {Browse ‘empty list’}
    end
Functions over lists

- Compute the function \{Pascal N\}
- Takes an integer N, and returns the Nth row of a Pascal triangle as a list
  1. For row 1, the result is [1]
  2. For row N, shift to left row N-1 and shift to the right row N-1
  3. Align and add the shifted rows element-wise to get row N

\[
\begin{align*}
  &\begin{array}{c}
    1 \\
    1 & 2 & 1 \\
  \end{array} \\
  &\left[\begin{array}{c}
    0 \\
    1 & 3 & 3 & 1 \\
  \end{array}\right] \\
  &\text{Shift right } [0 1 3 3 1] \\
  &\text{Shift left } [1 3 3 1 0]
\end{align*}
\]
declare
fun {Pascal N}
  if N==1 then [1]
  else
    {AddList
      {ShiftLeft {Pascal N-1}}
      {ShiftRight {Pascal N-1}}}
  end
end
Functions over lists (3)

fun {ShiftLeft L}  
  case L of H|T then 
    H|{ShiftLeft T} 
  else [0] 
  end 
end 

fun {ShiftRight L}  0|L end 

fun {AddList L1 L2}  
  case L1 of H1|T1 then 
    case L2 of H2|T2 then 
      H1+H2|{AddList T1 T2} 
    end 
  else nil end 
end
Top-down program development

• Understand how to solve the problem by hand
• Try to solve the task by decomposing it to simpler tasks
• Devise the main function (main task) in terms of suitable auxiliary functions (subtasks) that simplify the solution (ShiftLeft, ShiftRight and AddList)
• Complete the solution by writing the auxiliary functions
Is your program correct?

• “A program is correct when it does what we would like it to do”
• In general we need to reason about the program:
  • **Semantics for the language**: a precise model of the operations of the programming language
  • **Program specification**: a definition of the output in terms of the input (usually a mathematical function or relation)
  • Use mathematical techniques to reason about the program, using programming language semantics
Mathematical induction

• Select one or more inputs to the function
• Show the program is correct for the *simple cases* (base cases)
• Show that if the program is correct for a *given case*, it is then correct for the *next case*.
• For natural numbers, the base case is either 0 or 1, and for any number n the next case is n+1
• For lists, the base case is nil, or a list with one or a few elements, and for any list T the next case is H|T
Correctness of factorial

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

\[ 1 \times 2 \times \cdots \times (n - 1) \times n \]

\[ \text{Fact}(n-1) \]

• Base Case N=0: \{Fact 0\} returns 1
• Inductive Case N>0: \{Fact N\} returns N*{Fact N-1} assume \{Fact N-1\} is correct, from the spec we see that \{Fact N\} is N*{Fact N-1}
Complexity

• Pascal runs very slow, try \{Pascal 24\}
• \{Pascal 20\} calls: \{Pascal 19\} twice, \{Pascal 18\} four times, \{Pascal 17\} eight times, ..., \{Pascal 1\} $2^{19}$ times
• Execution time of a program up to a constant factor is called the program’s \textit{time complexity}.
• Time complexity of \{Pascal N\} is proportional to $2^N$ (exponential)
• Programs with exponential time complexity are impractical

```pascal
declare
fun {Pascal N}
  if N==1 then [1]
  else
    {AddList
     {ShiftLeft \{Pascal N-1\}}
     {ShiftRight \{Pascal N-1\}}}
  end
end
```

C. Varela; Adapted w. permission from S. Haridi and P. Van Roy
Faster Pascal

• Introduce a local variable L
• Compute \{FastPascal N-1\} only once
• Try with 30 rows.
• FastPascal is called N times, each time a list on the average of size N/2 is processed
• The time complexity is proportional to N^2 (polynomial)
• Low order polynomial programs are practical.

```fun \{FastPascal N\}
  if N==1 then [1]
  else
    local L in
    L=\{FastPascal N-1\}
    \{AddList \{ShiftLeft L\} \{ShiftRight L\}\}
    end
  end
end```
Lazy evaluation

- The functions written so far are evaluated eagerly (as soon as they are called)
- Another way is lazy evaluation where a computation is done only when the result is needed

- Calculates the infinite list:
  \[ 0 \mid 1 \mid 2 \mid 3 \mid \ldots \]

```
declare
fun lazy {Ints N}
    N|{Ints N+1}
end
```
Lazy evaluation (2)

• Write a function that computes as many rows of Pascal’s triangle as needed
• We do not know how many beforehand
• A function is lazy if it is evaluated only when its result is needed
• The function PascalList is evaluated when needed

fun lazy {PascalList Row}
  Row | {PascalList}
    {AddList}
    {ShiftLeft Row}
    {ShiftRight Row}}
end
Lazy evaluation (3)

- Lazy evaluation will avoid redoing work if you decide first you need the 10th row and later the 11th row.
- The function continues where it left off.

```declare
L = {PascalList [1]}
{Browse L}
{Browse L.1}
{Browse L.2.1}
```

```
L<Future>
[1]
[1 1]
```
Exercises

26. Lambda Calculus Handout Exercise 11.
27. Lambda Calculus Handout Exercise 12.
28. Define Add in Oz using the Zero and Succ functions representing numbers in the lambda-calculus:

```plaintext
Zero = fun {$ X} X end
Succ = fun {$ N} fun {$ X} N end
```

29. Prove that Add is correct using induction.

30. Prove the correctness of AddList and ShiftLeft using induction.

31. VRH Exercise 1.18.5.