Introduction to Programming Concepts (VRH 1.1-1.8)

Carlos Varela
RPI
September 24, 2007

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

\[
\text{declare} \quad V = 9999*9999
\]

• 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:

\[
\text{Fact} = \begin{cases} 
1 & \text{if } N = 0 \\
N \times \text{Fact}(N-1) & \text{if } N > 0 
\end{cases}
\]

• Start with the mathematical definition

\[
\text{Fact}(N) = \frac{n!}{r!(n-r)!}
\]

• Example of functional abstraction

Combining functions

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

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

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:
• The empty list is written nil
• Lists are created by means of \( \text{cons} \)

\[
\text{H} = \text{cons}(1 \text{ \text{cons} } 2 \text{ \text{cons} } 3 \text{ \text{cons} } 4 \text{ nil})
\]

\[
(\text{Browse } \text{H}) \quad \% \text{ This will show } [1 \text{ \text{cons} } 2 \text{ \text{cons} } 3 \text{ \text{cons} } 4 \text{ nil}]
\]
Lists (2)

- 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]
```

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) end
```

Functions over lists

- Compute the function \( \{Pascal\ N\} \)
- Takes an integer \( N \), and returns the \( N \)th 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 \( N \)

```
Pascal N
```

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

```
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} 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 simplifies 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

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

- 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\(^{th}\) times
  - Execution time of a program up to a constant factor is called the program’s time complexity.
  - Time complexity of \(\{Pascal N\}\) is proportional to \(2^N\) (exponential)
  - Programs with exponential time complexity are impractical

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.

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 results is needed
  - Calculates the infinite list: \(0\ |\ 1\ |\ 2\ |\ 3\ |\ ...
```
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)
end

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

Exercises

32. Define `Add` in Oz using the Zero and Succ functions representing numbers in the lambda-calculus.
33. Prove that `Add` is correct using induction.
34. *Prove the correctness of `AddList` and `ShiftLeft` using induction.
35. *VRH Exercise 1.18.5.