Introduction to Programming Concepts (VRH Chapter 1)

Carlos Varela

RPI

February 24, 2011

Adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

Introduction

- An introduction to programming concepts
- Declarative variables
- 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

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

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

- Fact is declared in the environment
- Try large factorial {Browse {Fact 100}}

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n$$

$$0! = 1$$

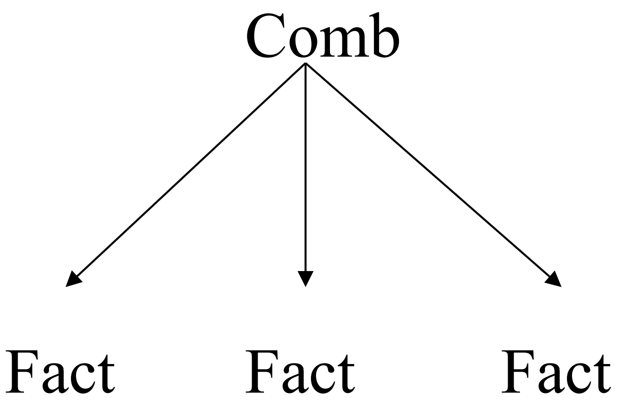
$$n! = n \times (n-1)! \text{ if } n > 0$$

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
```


Comb

Fact

Fact

Fact

- 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)

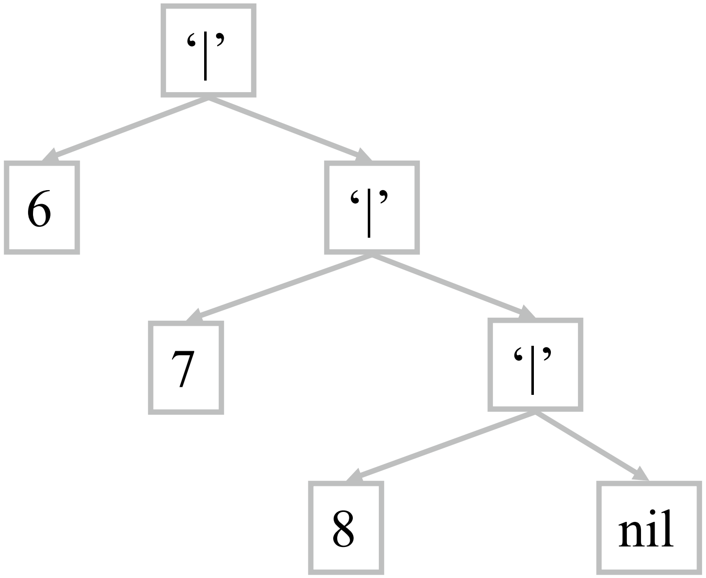
```
declare
H=1
T = [2 3 4 5]
{Browse H|T}  % This will show [1 2 3 4 5]
```

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

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

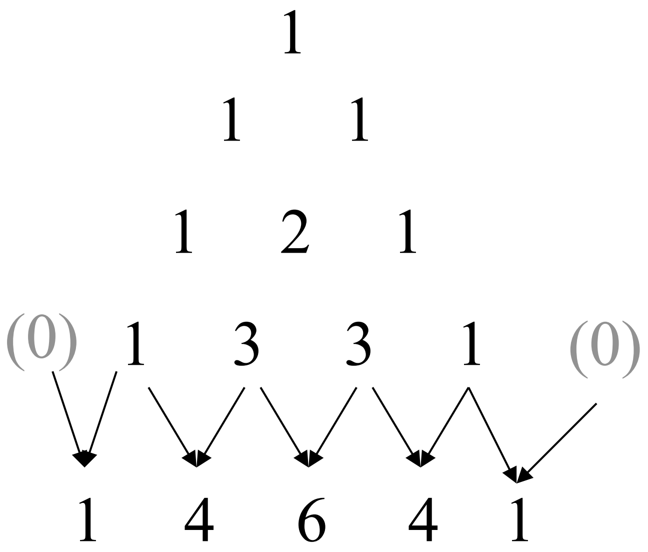
- Another way to take a list apart is by use of pattern matching with a case instruction

```
case L of H|T then {Browse H} {Browse T}
        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



1

1 1

1 2 1

(0) 1 3 3 1 (0)

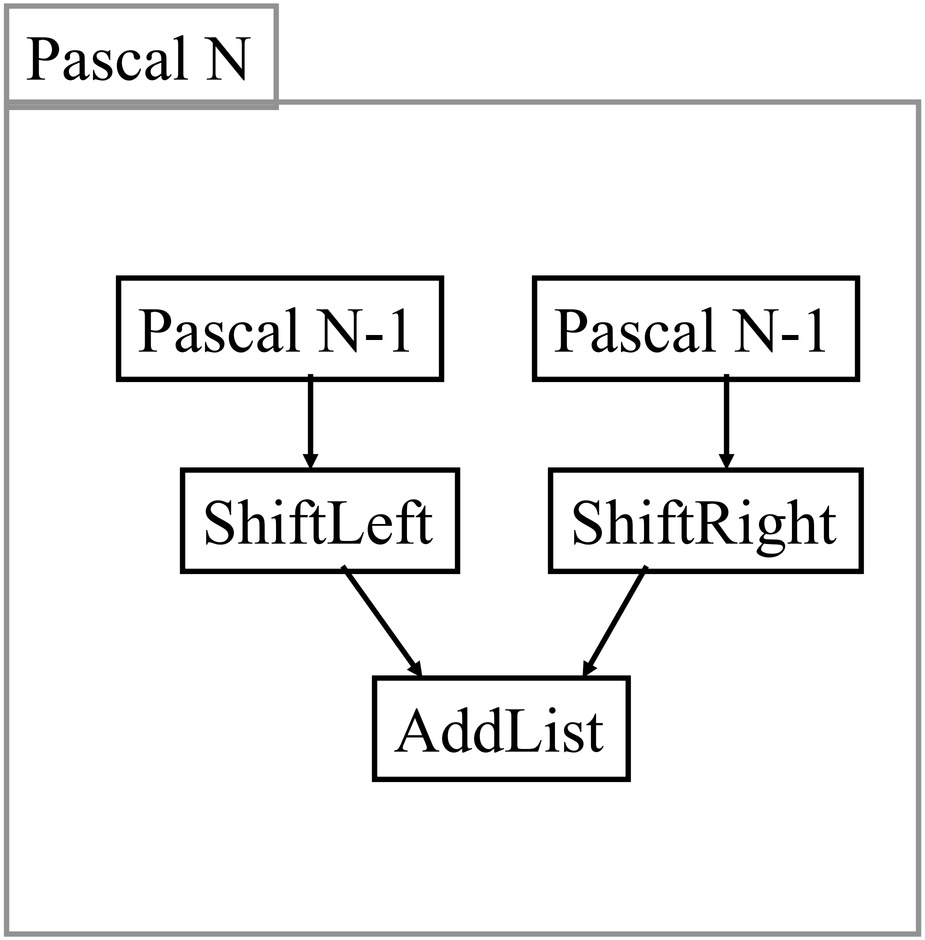
1 4 6 4 1

Shift right [0 1 3 3 1]

Shift left [1 3 3 1 0]

Functions over lists (2)

```
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
- Test your program bottom-up: auxiliary functions first.

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
```

$$\underbrace{1\times 2\times\cdots\times(n-1)}_{Fact(n-1)}\times n$$

- 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 *time complexity*.
- Time complexity of {Pascal N} is proportional to 2^N (exponential)
- Programs with exponential time complexity are impractical

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

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 | 1 | 2 | 3 | ...

```
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]
```

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
```

```
          1                                1
       1     1                          1     1
     1    2    1                      1    0    1
   1    3    3    1                 1    1    1    1
 1    4    6    4    1            1    0    0    0    1
```

Higher-order programming

```
fun {GenericPascal Op N}
  if N==1 then [1]
  else L in L = {GenericPascal Op N-1}
    {OpList  Op {ShiftLeft L} {ShiftRight L}}
  end
end
fun {OpList Op L1 L2}
     case L1 of H1|T1 then
              case L2 of H2|T2 then
                   {Op H1 H2}|{OpList Op T1 T2}
              end
     else nil end
end
```

```
fun {Add N1 N2} N1+N2 end
fun {Xor N1 N2}
     if N1==N2 then 0 else 1 end
end

fun {Pascal N} {GenericPascal Add N} end
fun {ParityPascal N}
     {GenericPascal Xor N}
end
```

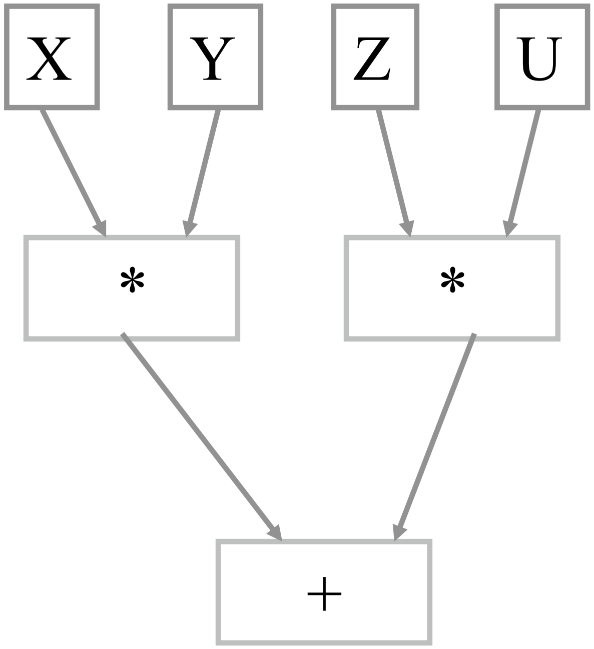
Concurrency

- 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

```
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*

```
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

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

```
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

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

Classes (2)

- Here is a class with two operations: Bump and Read

```
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

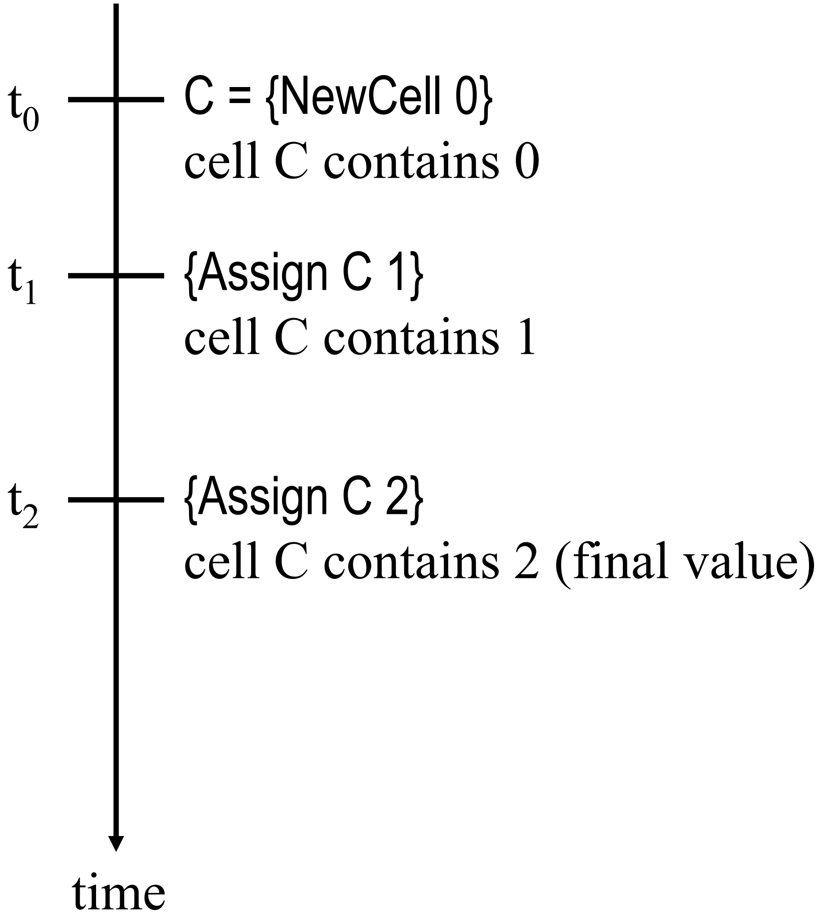
- In object-oriented programming the idea of objects and classes is pushed farther
- Classes keep the basic properties of:
  - State encapsulation
  - Object factories
- Classes are extended with more sophisticated properties:
  - They have *multiple* operations (called *methods*)
  - They can be defined 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

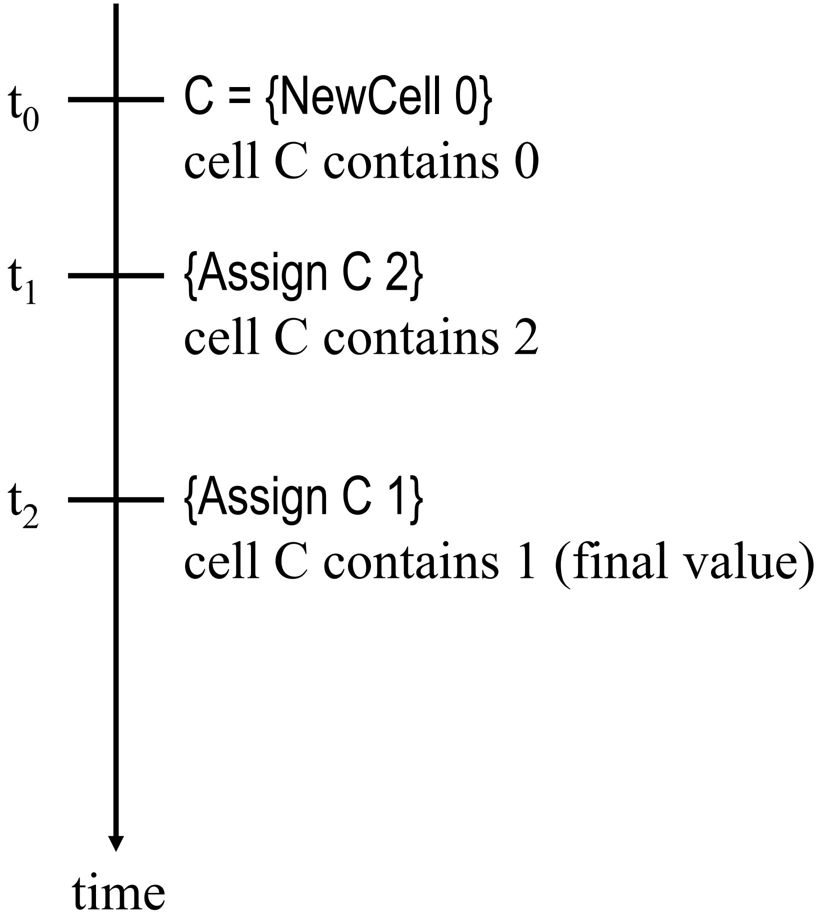
t_2 {Assign C 2}

cell C contains 2 (final value)

time

Nondeterminism (3)

```
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 2}
cell C contains 2

t_2 {Assign C 1}
cell C contains 1 (final value)

time

Nondeterminism (4)

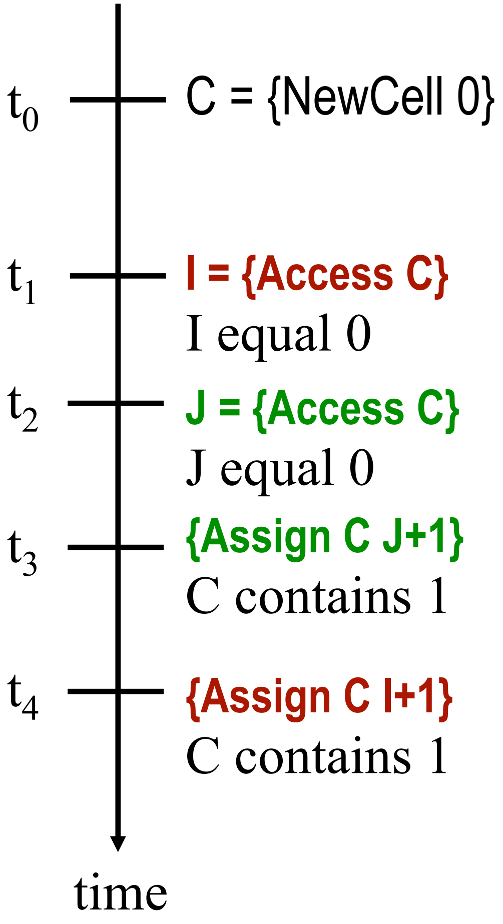
```
declare
C = {NewCell 0}

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?

Nondeterminism (5)

- 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
```



t_0 C = {NewCell 0}

t_1 I = {Access C}
I equal 0

t_2 J = {Access C}
J equal 0

t_3 {Assign C J+1}
C contains 1

t_4 {Assign C I+1}
C contains 1

time

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)

```
declare
L = {NewLock}
```

```
lock L then
  sequence of ops 1
end
```

Thread 1

```
lock L then
  sequence of ops 2
end
```

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 Kth row)
  3. if N is less or equal to K retrieve the Nth row from S
  4. Otherwise, compute the rows numbered K +1 to N, and store them in S
  5. Return the Nth 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

24. Lambda Calculus Chapter Exercise 7.
25. Lambda Calculus Chapter Exercise 9.
26. Lambda Calculus Chapter Exercise 11.
27. Lambda Calculus Chapter Exercise 12.
28. Define Add in Oz using the Zero and Succ functions representing numbers in the lambda-calculus:

    ```
    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.

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).