Declarative Concurrency

Annalise Tarhan

May 10, 2020

1 Thread Semantics

```
local B in
     thread B=true end
     thread B=false end
     if B then {Browse yes} end
end
```

1.1 Enumerate all possible executions of the previous statement.

1={thread B=true end}, 2={thread B=false end}, and 3={if B then {Browse yes} end}. The possible executions are 123, 132, 213, 231. 3 cannot be executed first because it is not ready until B is determined.

1.2 Some of these executions cause the program to terminate abnormally. Make a small change to avoid these terminations.

```
thread if B then {Browse yes} end end
```

2 Threads and Garbage Collection

```
proc {B _}
      {Wait _}
end

proc {A _}
      Collectible={NewDictionary}
in
      {B Collectible}
end
```

2.1 After the call $\{A\}$ is done, will the memory occupied by *Collectible* be recovered?

Yes, it will. After the procedure ends, nothing holds a reference to *Collectible*. B holds a reference to it as long as it is unbound, since $\{Wait\}$ waits until it is, but by the time B is called, it has been bound to a Dictionary. This can be verified by calling $\{Browse\ \_\}$ within B, which displays $< Dict >$.

3 Concurrent Fibonacci

```
fun {Fib X}
      if X=<2 then 1
      else {Fib X-1}+{Fib X-2} end
end
```

3.1 Compare the above function with the concurrent definition. How much faster is the sequential definition? How many threads are created by the concurrent call {Fib N} as a function of N?

The sequential function is roughly five times faster than the concurrent function. The number of threads created is approximately the same as the result of $\{Fib\ N\}$.

4 Order-Determining Concurrency

```
declare A B C D in
thread D=C+1 end
thread C=B+1 end
thread A=1 end
thread B=A+1 end
{Browse D}
```

4.1 In what order are the threads created? In what order are the additions done? What is the final result?

The threads are created sequentially. The additions, however, are only performed when the addends are determined. *A* is the first variable to be bound, which means that *B* can be calculated. Once *B* is calculated, *C* can be calculated, and finally *D*. The result is 4.

```
declare A B C D in
A=1
B=A+1
C=B+1
D=C+1
{Browse D}
```

4.2 In this version, there is only one thread. In what order are the additions done? What is the final result? What do you conclude?

In this version, the order of calculations and the final result are the same as in the concurrent version. This implies that calculating concurrently does not change either the order of calculations or their result, it simply allows for more flexibility when writing the code than the sequential model does.

5 The *Wait* Operation

```
proc {Wait X}
     if X==unit then skip else skip end
end
```

5.1 Explain why the $\{Wait\ X\}$ operation could be defined this way.

This definition works because the correct behavior of the *Wait* operation is to suspend until X is determined, and that is exactly what both the if statement and the entailment check do. The *if* statement suspends until the activation condition is true, in this case when the result of the entailment check has been determined. The entailment check suspends until it can determine whether the two values are equal, which it can't do until X is bound. It doesn't matter what is on the other side of the equality operator.

6 Thread Scheduling

6.1 If the technique of skipping over already-calculated elements of a stream is used, the sum of the elements in the integer stream is much smaller than the actual sum of the integers in the stream. Why? Explain the result in terms of thread scheduling.

Since the producing stream and the consuming stream run in separate threads, the scheduler alternates between them. Assuming the scheduler is efficiently dividing time between them, it lets each stream do more than one operation at a time, so the producer generates multiple values during its allotted time. The technique of skipping over already-calculated elements means the consuming stream will only see the most recent value generated by the producing stream, so most of the values created by the producer won't be included in the sum. This will make the sum much smaller than it would have been otherwise.

7 Programmed Triggers Using Higher-Order Programming

7.1 Rewrite the programmed trigger example that uses concurrency and dataflow variables to use higher-order programming. The producer should pass a zero-argument function F to the consumer, which the consumer calls whenever it needs an element. It should return a pair $X\#F2$ where X is the next stream element and $F2$ is a function that has the same behavior as F.

```
fun {Generate N}
     F=fun {$} {Generate N+1} end
in
     N#F
end

fun {Sum F A Limit}
     if Limit>=0 then
          case {F} of X#F2 then
               {Sum F2 A+X Limit-1}
          end
     else A end
end
```

8 Dataflow Behavior in a Concurrent Setting

```
fun {Filter In F}
     case In
     of X|In2 then
          if {F X} then X|{Filter In2 F}
          else {Filter In2 F} end
     else
          nil
     end
end
```

8.1 What happens when the following is executed?

```
declare A
{Show {Filter [5 1 A 4 0] fun {$ X} X>2 end}}
```

When *Filter* reaches the unbound variable identifier, it suspends indefinitely and doesn't return anything.

8.2 What happens when the following is executed? What is displayed and why? Is the *Filter* function deterministic? Why or why not?

```
declare Out A
thread Out={Filter [5 1 A 4 0] fun {$ X} X>2 end} end
{Show Out}
```

Filter still can't proceed past *A*, since it isn't determined, but because it is calculating in its own thread, there are different possible results. *Show* could display _, representing an unbound variable identifier, or 5|_, representing a list pair with an unbound tail. Either is possible, because the scheduler could switch to the thread *Show* is in either before *Filter* is called or after it has started to bind *Out*.
The *Filter* function itself is deterministic because it always returns the same result for the same arguments. However, the Mozart system uses a hardware timer, which makes the scheduler nondeterministic because preemption could occur at different instants.

8.3 What happens when the following is executed?

```
declare Out A
thread Out={Filter [5 1 A 4 0] fun {$ X} X>2 end} end
{Delay 1000}
{Show Out}
```

In this case, the 1000 millisecond delay is enough to ensure that *Out*'s thread will calculate as much as it can before it is shown. It still suspends at *A*, for a final value of 5|_.

8.4 What happens when the following is executed? What is displayed and why?

```
declare Out A
thread Out={Filter [5 1 A 4 0] fun {$ X} X>2 end} end
thread A=6 end
{Delay 1000}
{Show Out}
```

There is only one possible result, for the same reason as the previous section. Here, *A* is eventually bound, so by the time it is shown, the value of *Out* is [5 6 4].

9 Digital Logic Simulation

9.1 Design a circuit to add n-bit numbers using a chain of full adders.

```
fun {AddHelper A B Carry Result}
    case A
    of nil then
        if Carry==0
        then Result
        else Carry|Result
        end
    [] X|Xr then Cy Sm in
        {FullAdder A.1 B.1 Carry Cy Sm}
        {AddHelper A.2 B.2 Cy Sm|Result}
    end
end

fun {Add A B}
    {AddHelper {Reverse A} {Reverse B} 0 nil}
end
```

10 Basics of Laziness

```
fun lazy {Three} {Delay 1000} 3 end
```

10.1 Calculating $\{Three\}$+0 three times in succession takes three seconds. Why is this, if $Three$ is supposed to be lazy? Shouldn't its result be calculated only once?

Lazy doesn't mean it calculates only once, it means it waits to calculate until its result is needed. All of the calls to $\{Three\}$ are in the same thread, so each call doesn't begin until the previous call has finished, which results in an additional one second delay.

11 Laziness and Concurrency

```
fun lazy {MakeX} {Browse x} {Delay 3000} 1 end
fun lazy {MakeY} {Browse y} {Delay 6000} 2 end
fun lazy {MakeZ} {Browse z} {Delay 9000} 3 end

X={MakeX}
Y={MakeY}
Z={MakeZ}

{Browse (X+Y)+Z}
```

11.1 This displays x and y immediately, z after six seconds, and the result 4 after fifteen seconds. Why?

When I run this program, that is not the result I get. x is displayed immediately. y after three seconds, and z after nine. Since everything is running sequentially, this is the behavior I expect.

11.2 What happens if $(X+Y)+Z$ is replaced by $X+(Y+Z)$?

This changes the order of the calculations, since Y and Z are calculated before X, but not the total calculation time, which is still 15 seconds.

11.3 What happens if $(X + Y) + Z$ is replaced by thread $X + Y$ end + Z?

Interestingly, this still doesn't change anything. I would expect that Z would be calculated concurrently, since the $X + Y$ is happening in its own thread. However, since that clearly isn't what happened, I would guess that Z is not actually calculated until after $X + Y$ is resolved because the first addend must be resolved before the second is needed. That doesn't seem to be true, though, because if I change it to $Z+$ thread $(X + Y)$ end, it does calculate concurrently and only takes 9 seconds.

11.4 How would you program the addition of n integers $i_1, ..., i_n$, given that integer i_j only appears after t_j ms, so that the final result appears the quickest?

I would throw everything in its own thread and hope for the best, since the results of the previous sections don't seem to align either with my own intuition or with what the textbook authors imply that they should be.

12 Laziness and Incrementality

12.1 The output stream from a producer/consumer pair implemented using either concurrency or laziness could appear incrementally. What is the difference?

In both examples, the producer and the consumer and working with the same stream. The difference is when the producer is triggered to produce more elements to bind to the stream. In the concurrent example, it produces whenever the scheduler gives its thread a time slice and it continues until its time runs out or it reaches the limit. In the lazy example, the producer waits until the consumer tries to access the next element, then binds it. In both cases, the producer binds the output stream incrementally, so they can both be displayed incrementally.

12.2 What happens if you use both concurrency and laziness at the same time?

The stream can still appear incrementally and does so at the same slow speed as the lazy example. In fact, adding concurrency to the lazy example changes very little, since laziness is implemented using $ByNeed$, and $ByNeed$ calls the procedure in its own thread.

13 Laziness and Monolithic Functions

```
fun lazy {Reverse1 S}
     fun {Rev S R}
          case S of nil then R
          [] X|S2 then {Rev S2 X|R} end
     end
in {Rev S nil} end

fun lazy {Reverse2 S}
     fun lazy {Rev S R}
          case S of nil then R
          [] X|S2 then {Rev S2 X|R} end
     end
in {Rev S nil} end
```

13.1 What is the difference in behavior between {Reverse1 [a b c]} and {Reverse2 [a b c]}?

There is no difference. They calculate the same result, they calculate it all at once, and they do it at the same time, when the result is touched. The first version makes the recursive call and executes immediately, while the second version takes the intermediate step of suspending each call first before executing it.

13.2 Compare the execution efficiency of the two definitions. Which definition would you use in a lazy program?

The first version is much more efficient. Since the programmer would know that all of the intermediate recursive calls would need to happen immediately in order to get even the first result, it would make no sense to add the extra overhead of suspending each call. A monolithic function should behave monolithically, even if the function itself is lazy.

14 Laziness and Iterative Computation

14.1 Consider a straightforward lazy version of *Append* without dataflow variables. Is it iterative? Why or why not?

In the lazy version, the dataflow variable is replaced by a function call in the trigger store. Since a function is iterative if its stack size is bounded by a constant, this version is iterative for the same reason as the version with dataflow variables: its stack size doesn't grow.

15 Performance of Laziness

15.1 Choose some declarative programs and rewrite them to make them lazy. Compare their performance with the originals.

```
fun lazy {CubeRootLazy X}
      fun lazy {Add A B} A+B end
      fun lazy {Sub A B} A-B end
      fun lazy {Div A B} A/B end
      fun lazy {Mul A B} A*B end
      fun lazy {Cube A} A*A*A end
      fun lazy {Improve Guess}
            {Div {Add {Div X {Mul Guess Guess}}
            {Mul 2.0 Guess}} 3.0}
      end
      fun lazy {Abs Y}
            if Y<0.0 then ~Y else Y end
      end
      fun lazy {GoodEnough Guess}
            {Div {Abs {Sub X {Cube Guess}}} X} < 0.00001
      end
      fun lazy {CubeRootIter Guess}
            if {GoodEnough Guess} then Guess
            else {CubeRootIter {Improve Guess}}
            end
      end
      Guess=1.0
in
      {CubeRootIter Guess}
end
```

The lazy version of *CubeRoot*, even with all of its lazy arithmetic, seems to be just as fast as the original.

```
fun lazy {AppendIterLazy L1 L2}
     fun lazy {Reverse Original Reversed}
          case Original
          of nil then Reversed
          [] Head|Tail then {Reverse Tail Head|Reversed}
          end
     end
     fun lazy {AppendReversed ReversedPart SecondPart}
          case  ReversedPart
          of nil then SecondPart
          [] Head|Tail
               then {AppendReversed Tail Head|SecondPart}
          end
     end
in
     {AppendReversed {Reverse L1 nil} L2}
end
```

Lazy *AppendIter* didn't fare so well. With a large input, it took over ten times as long as the original.

16 By-Need Execution

16.1 Define an operation that requests the calculation of X but that does not wait.

```
thread if X==unit then skip else skip end end
```

17 Hamming Problem

17.1 Write a program that solves the Hamming problem. For any given n and k, it should return the first n integers of the form $p_1^{a_1}p_2^{a_2}...p_k^{a_k}$ with $a_1, a_2, ..., a_k \geq 0$.

```
fun {Hamming NumIntegers NumPrimes}
     fun lazy {Generate N}
          N|{Generate N+1}
     end
     fun lazy {Filter Xs F}
          case Xs
          of nil then nil
          [] X|Xr andthen {F X} then X|{Filter Xr F}
          [] _|Xr then {Filter Xr F}
          end
     end
     fun {Sieve Xs N}
          if N==0 then nil else
          case Xs of X|Xr then
               Ys in
               Ys={Filter Xr fun {$ Y} Y mod X \= 0 end}
               X|{Sieve Ys N-1}
               end
          end
     end
     fun {GetPrimes}
          {Sieve {Generate 2} NumPrimes}
     end
     fun lazy {Times Prime Stream}
          case Stream of H|T then
          Prime*H|{Times Prime T} end
     end
     fun lazy {MergeTwo Xs Ys}
          case Xs#Ys of (X|Xr)#(Y|Yr) then
               if X<Y then X|{MergeTwo Xr Ys}
               elseif X>Y then Y|{MergeTwo Xs Yr}
               else X|{MergeTwo Xr Yr}
               end
          end
     end
     fun lazy {MergeAll Primes}
          case Primes
          of A|B|nil then {MergeTwo A B}
          [] A|B then {MergeTwo A {MergeAll B}}
          [] A then A
          end
     end
     proc {Touch N H}
          if N>0 then {Touch N-1 H.2} else skip end
     end
     Primes H
in
     Primes={GetPrimes}
     H=1|{MergeAll {Map Primes fun {$ N} {Times N H} end}}
     {Touch NumIntegers H}
     H
end
```

18 Concurrency and Exceptions

```
proc {TryFinally S1 S2}
B Y in
     try {S1} B=false catch X then B=true Y=X end
     {S2}
     if B then raise Y end end
end
```

18.1 Based on the previous control abstraction, determine the different possible results of the following program:

```
local U=1 V=2 in
     {TryFinally
     proc {$}
          thread
               {TryFinally proc {$} U=V end
                           proc {$} {Browse bing} end}
          end
     end
     proc {$} {Browse bong} end}
end
```

Because the first statement in the outer *TryFinally* call is wrapped in *thread...end*, it is unknown whether it would execute before or after the second statement inside the outer *TryFinally* call. If it executes first, the inner *try* statement catches the unification failure exception, sets *B* to true, and displays *bing*. Then, the outer *TryFinally* will also catch the exception, set *B* to true, then display *bong*.

If the thread isn't executed first, the outer *TryFinally* statement will continue, set *B* to false, and display *bong*. Only later will the inner *TryFinally* statement execute, setting *B* to true and displaying *bing*. This would result in an uncaught exception. It is also possible for interleaving to occur, which could reverse the order of *bing* and *bong*.

In any case, there are two possible results, *bing bong* and *bong bing*. Including the possibility of interleaving, there are four possible executions.

19 Limitations of Declarative Concurrency

19.1 Declarative concurrency cannot model client/server applications, because the server cannot read commands from more than one client. However, the declarative $Merge$ function reads from three input streams to generate one output stream. How can this be?

In the client/server model, each of the clients is independent, meaning they operate concurrently. This leads to non-determinism. All of the streams $Merge$ operates on are created synchronously, so the operations are deterministic.

20 Worst-Case Bounds with Laziness (advanced)

20.1 Investigate how to write a queue with a constant worst-case bound.

One way to write a queue with a constant worst case bound is to start with the given queue, which has an amortized constant bound, and extend it with a schedule. The schedule works by holding an unevaluated function call which will eventually perform the same rotation the original queue does.

When an item is added to or removed from the queue, a pseudo-constructor is called which maintains the invarient that the size of the schedule is equal to the size of the front part of the queue minus the size of the back part. Usually, this will just evaluate and remove one node from the schedule. Since the front of the queue contains a reference to the same node, it will have access to the evaluated value. When the schedule is empty, it will force a rotation. This takes the front of the queue and the back of the queue, and begins the same rotation process as the original. When it is finished, the queue will consist of a front part and schedule with the same content, a mostly unevaluated stream, and an empty back part.

Typical operations, adding and removing elements from the queue when the schedule isn't empty, consist of adding or removing the element, plus the schedule evaluating a singe node. Since the 'rotation' also happens in constant time, the whole queue has a constant worst-case bound.

This answer is based on the paper referenced in the book:
https://www.cs.cmu.edu/ rwh/theses/okasaki.pdf

21 Controlling Concurrency (research project)

21.1 The declarative concurrent model gives three primitive operations that affect execution order without changing the results of a computation: sequential composition, lazy execution, and concurrency. These operations can be used to tune the order in which a program accepts input and gives results, e.g., to be more or less incremental. This is a good example of separation of concerns. Investigate this topic further.

Separation of concerns is the principle that software should be structured such that each module is as decoupled from others as possible. Changes to one module should not affect others. In this case, the separation is temporal, since sequential, lazy, and concurrent executions differ only in when calculations are performed.

Sequential composition results in inflexible code. Each operation will be performed in exactly the same order, requiring inputs to be given at a certain time and in a certain order. Results will be given in the same rigid way. Operations are performed monolithically, so this is the least incremental and most tightly coupled option.

Lazy execution decouples parts of the same operation. Calculations only occur when their results are needed, so each operation is dependent on the function that depends on its result. Results will be given exactly when they are needed, so this approch is completely incremental, but somewhat tightly coupled between the lazy function and the function that uses its results.

Concurrency is the least tightly coupled option. Each concurrent operation happens in its own thread and as such is completely independent from whatever else is happening in the program. If it relies on inputs, it will pick up calculating whenever it recieves them. Results are given as soon as they are calculated. Concurrency is not inherently incremental, but it plays an essential role in lazy execution. When the result of lazy execution is needed, it triggers the creation of a thread to calculate the result.

21.2 Are these three operations complete? That is, can all possible partial execution orders be specified with them?

Sequential composition is totally ordered, since there is only one possible execution order. Lazy execution is also totally ordered. The execution order is less straightforward, since operations will only complete as much as is needed, but the calculations will always occur in the same order. Concurrency allows for many partial orders based on the permutations of non-causally ordered steps. However, if there are a finite number of steps, there are a finite number of permutations. Therefore, for a program without the possibility of infinite recursion, all three operations are complete. It is only for a concurrent program that could run indefinitely that the number of possible partial execution orders could not be specified.

21.3 What is the relationship with reduction strategies in the λ calculus, e.g., applicative order reduction, normal order reduction?

In lambda calculus, applicative order reduction reduces the arguments first, then applies the function. Normal order reduction does the opposite, applying the function first and then evaluating the arguments. These strategies parallel eager and lazy execution. Evaluating the arguments first, as applicative order reduction does, is comparable to eager execution, where expressions are evaluated as soon as they are given arguments. Lazy evaluation only applies the function to the arguments when the result is needed, which is similar to the strategy of normal order reduction.

21.4 Are dataflow or single-assignment variables essential?

For sequential composition, no they are not. Strict functional programming languages, such as Scheme, do not use dataflow variables. There are also lazy functional languages, such as Haskell, that don't use them, so even though Oz uses them to implement laziness, they are not essential for laziness in general. For declarative concurrency, however, they are essential. They are the mechanism by which threads synchronize and communicate.