Aluno: Carlos Victor Martins da Silva

Matrícula: 476516

| **Questões feitas** | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1.** | **2.** | **3.** | **4.** | **5.** | **6.** | **7.** | **8.** | **10.** | **11.** |
| a) X b) X | X | X | X | X | X | X | a) X b) X c) X d) X | X | X |
| **12.** | | **13.** | | **14.** | | **18.** | | **19.** | |
| X | | X | | X | | X | | X | |

1. **Thread semantics.** Consider the following variation of the statement used in Section 4.1.3 to illustrate thread semantics:

local B in

thread B = true end

thread B = false end

if B then {Browse yes} end

end

For this exercise, do the following:

(a) Enumerate all possible executions of this statement.

Resposta:

| **Legenda** | **Possíveis ordens de execução** |
| --- | --- |
| a. local B in ... end  b. thread … end (linha 2)  c. thread … end (linha 3)  d. if B then … end  e. {Browse yes}  f. B = True  g. B = False | 1. a,b,c,f,d,e,g  2. a,b,c,g,d,e,f  3. a,b,f,c,d,e,g  4. a,b,c,f,d,g  5. a,b,c,g,d,f  6. a,b,f,c,d,g  7. a,b,c,f,g  8. a,b,c,g,f  9. a,b,f,c,g |

(b) Some of these executions cause the program to terminate abnormally. Make a small change to the program to avoid these abnormal terminations.

Resposta:

| A única forma de garantir a execução do programa é adicionar a possibilidade de uma thread redefinir o valor de B após já ter sido definido por outra, ou seja, utilizar uma Cell para armazenar o valor booleano.  local B in  B = {NewCell null}  thread B := true end  thread B := false end  if @B then  {Browse yes}  end  end |
| --- |

2. **Threads and garbage collection.** This exercise examines how garbage collection behaves with threads and dataflow variables. Consider the following program:

proc {B \_}

{Wait \_}

end

proc {A}

Collectible = {NewDictionary}

in

{B Collectible}

end

After the call {A} is done, will Collectible become garbage? That is, will the memory occupied by Collectible be recovered? Give an answer by thinking about the semantics. Verify that the Mozart system behaves in this way

Resposta:

| O procedimento Wait bloqueia a execução de um outro procedimento ou função até que um argumento seja passado e o dicionário armazenado em Collectible é passado como um argumento na chamada do procedimento B tendo ele já sido referenciado e armazenado antes da chamada de B, ou seja, ele está sujeito a ser limpo da memória já que o procedimento Wait trava a execução de B. |
| --- |

3. **Concurrent Fibonacci.** Consider the following sequential definition of the Fibonacci function:

fun {Fib X}

if X =< 2 then 1

else

{Fib X - 1} + {Fib X - 2}

end

end

and compare it with the concurrent definition given in Section 4.2.3. Run both on the Mozart system and compare their performance. How much faster is the sequential definition? How many threads are created by the concurrent call {Fib N} as a function of N?

Resposta:

| % O programa utilizado para testes ao final fica desta forma:  declare Fib ConcurrentFib Timer  fun {Fib X}  if X =< 2 then 1  else {Fib X-1} + {Fib X-2} end  end  fun {ConcurrentFib X}  if X =< 2 then 1  else  thread {Fib X-1} end + thread {Fib X-2} end  end  end  % Para analisar o tempo gasto na execução das duas implementações da sequência de Fibonacci, utilizarei essa função que armazena o tempo inicial antes de fazer a execução da função ou procedimento passado no argumento Proc usando o argumento passado em Args e calcular ao final a diferença de tempo entre o ponto inicial e ponto final (final da execução).  fun {Timer Proc Args}  Start = {Time.time}  in  local T in  T = {Proc Args}  end    {Time.time} - Start  end  declare X = 40 % Argumento utilizado  {Browse {Timer Fib X}}  {Browse {Timer ConcurrentFib X}}  % Após várias execução notei que o tempo médio gasto entre a versão recursiva e concorrente da sequência de Fibonacci tem tempo médio parecido, no entanto, a versão concorrente pelo uso de threads, acaba, em algumas situações, gastando um tempo até 20% maior devido a necessidade de criar e encerrar threads durante a sua execução, algo que a versão recursiva faz utilizando a pilha de execução com maior eficiência.  O número de threads é similar ao número de chamadas a função Fib na recursão f(n) = f(n-1) + f(n-2), ou seja, o número de threads é exponencial da mesma forma que a função Fib. |
| --- |

4. **Order-determining concurrency.** Explain what happens when executing the following:

declare ABCD in

thread D = C + 1 end

thread C = B + 1 end

thread A = 1 end

thread B = A + 1 end

{Browse D}

In what order are the threads created? In what order are the additions done? What is the final result? Compare with the following:

declare ABCD in

A = 1

B = A + 1

C = B + 1

D = C + 1

{Browse D}

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

Resposta:

| As threads serão iniciadas segundo a ordem de declaração e as operações de soma serão feitas na ordem A->B->C->D pois o operador ‘+’ espera até que o valor da variável necessária para executar a operação esteja definida dentro do ambiente, assim, independente da ordem de execução, o resultado sempre será igual a 4. |
| --- |

5. **The** *Wait* **operation.** Explain why the {Wait X} operation could be defined as:

proc {Wait X}

if X == unit then skip else skip end

end

Use your understanding of the dataflow behavior of the if statement and == operation.

Resposta:

| A operação de verificação de igualdade feita por ‘==’ bloqueia a execução quando a expressão possui o mesmo tipo e que não possuem variáveis ligadas, o ‘if’ também tem esse mesmo comportamento. |
| --- |

6. **Thread scheduling.** Section 4.7.3 shows how to skip over already-calculated elements of a stream. If we use this technique to sum the elements of the integer stream in Section 4.3.1, the result is much smaller than 11249925000, which is the sum of the integers in the stream. Why is it so much smaller? Explain this result in terms of thread scheduling.

Resposta:

| A máquina virtual do ambiente Mozart executa as thread seguindo uma ordem de prioridade, dessa forma, evita com que haja mais produtores do que consumidores e vice-versa em execução, garantindo assim uma equidade no consumo de recursos e dados pelo dataflow. Quando o procedimento Skip é chamado dentro de um consumidor, o fluxo perde sua equidade gerando que alguns segmentos de dados gerados pelos produtos sejam pulados/esquecidos pelos consumidores, gerando uma superprodução. |
| --- |

7. **Programmed triggers using higher-order programming.** Programmed triggers can be implemented by using higher-order programming instead of concurrency and dataflow variables. The producer passes a zero-argument function F to the consumer. Whenever the consumer needs an element, it calls the function. This returns a pair X#F2 where X is the next stream element and F2 is a function that has the same behavior as F. Modify the example of Section 4.3.3 to use this technique.

Resposta:

| declare  proc {DGenerate N ?Xs}  NextXs  fun {NF}  {DGenerate N + 1 NextXs}  NextXs  end  in  Xs = N#NF  end  fun {DSum Xs ?A Limit}  if Limit > 0 then  X#NF = {Xs}  in  {DSum NF A + X Limit - 1}  else  A  end  end  local Xs S in  {DGenerate 0 Xs}  S = {DSum Xs.2 0 4096}  {Browse S}  end |
| --- |

8.**Dataflow behavior in a concurrent setting.** Consider the function {Filter In F}, which returns the elements of In for which the boolean function F returns true. Here is a possible definition of Filter:

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

Executing the following:

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

displays:

[5 4]

So *Filter* works as expected in the case of a sequential execution when all the input values are available. Let us now explore the dataflow behavior of *Filter*.

(a) What happens when we execute the following:

declare A

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

One of the list elements is a variable A that is not yet bound to a value. Remember that the case and if statements will suspend the thread in which they execute, until they can decide which alternative path to take.

Resposta:

| Graças ao uso do ‘if’, o código irá travar a execução esperando o resultado da função de comparação passada no 2° argumento. |
| --- |

(b) What happens when we execute the following:

declare Out A

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

{Show Out}

Remember that calling Show displays its argument as it exists at the instant of the call. Several possible results can be displayed; which and why? Is the Filter function deterministic? Why or why not?

Resposta:

| A função Filter tem o comportamento determinista pois, vincula seus resultados a variável Out. Uma vez que o procedimento Show é uma continuação porque está fora do modelo que mostra o resultado no momento de sua chamada, sendo assim, não-determinista observável. |
| --- |

(c) What happens when we execute the following:

declare Out A

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

{Delay 1000}

{Show Out}

Remember that the call {Delay N} suspends its thread for at least N milliseconds. During this time, other ready threads can be executed.

Resposta:

| Como a thread gera um paralelismo no flow de execução do programa, dessa forma, após o delay o resultado “5|\_” é exibido. |
| --- |

(d) What happens when we execute the following:

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}

What is displayed and why?

Resposta:

| [5 6 4] é exibido pela forma com que o operador ‘>’ aguarda até que A seja definido, permitindo assim ser comparado. |
| --- |

10. **Basics of laziness.** Consider the following program fragment:

fun lazy {Three} {Delay 1000} 3 end

Calculating {Three}+0 returns 3 after a 1000 millisecond delay. This is as expected, since the addition needs the result of {Three}. Now calculate {Three}+0 three times in succession. Each calculation waits 1000 milliseconds. How can this be, since Three is supposed to be lazy. Shouldn’t its result be calculated only once?

Resposta:

| A função Three será convertida para um procedimento na linguagem núcleo: (aproximação feita)  declare Three A B C  Three = proc {$ ?X}  {Delay 1000}  X=3  end  {ByNeed Three A} A + 0  {ByNeed Three B} B + 0  {ByNeed Three C} C + 0  Devido a necessidade de ser invocado por 3 vezes pelo ByNeed inserido para garantir que o resultado estivesse pronto para que a operação de soma (‘+’) pudesse ser feita, embora a função fosse lazy o seu resultado acabaria por ser calculado a cada operação feita. No modelo declarativo é possível adicionar um pequeno cache temporário que armazena o resultado de Three de forma a fazê-la ser executada por uma única vez ganhando assim evitando o delay de 1000ms forçado pelo procedimento. |
| --- |

11. **Laziness and concurrency I.** This exercise looks closer at the concurrent behavior of lazy execution. Execute the following:

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}

This displays x and y immediately, z after 6 seconds, and the result 6 after 15 seconds. Explain this behavior. What happens if (X+Y)+Z is replaced by X+(Y+Z) or by **thread** X+Y **end** + Z? Which form gives the final result the quickest? How would you program the addition of n integers i1, ..., in, given that integer ij only appears after tj milliseconds, so that the final result appears the quickest?

Resposta:

| - O que acontece se (X+Y)+Z é substituído por X+(Y+Z)?  A ordem com que as letras são mostradas no Browser do ambiente Oz passa “x, y, z” para “y, z ,x”. O motivo para isso é devido a formatação da operação dentro do argumento passado para o procedimento Browse, as funções MakeY e MakeZ têm prioridade maior tendo assim seus resultados calculados antecipadamente de MakeX.  - Ou caso fosse substituído por thread X+Y end + Z ?  Assim como no caso anterior, a ordem de prioridade muda e passa a ser imediato a necessidade dos resultados de MakeX e MakeY que estão contidos dentro da Thread. Sendo assim o valor de Z só é necessário 15 segundos após o processamento de X+Y ser feito.  - Qual a forma mais rápida de mostrar o resultado no final?  A forma mais rápida seria utilizando threads ao invés de funções lazy, considerando que as threads levam tempo máximo de até 1 segundo para serem iniciadas, dessa forma poderíamos ter **n** threads em execução que levariam **n** tempo para calcular e retornar os resultados para ser armazenado e somado ao final. Já quanto à ordem do resultado, isso é garantido pelo comportamento padrão do operador ‘+’ que aguarda que os dados a direita e esquerda estejam definidos dentro do ambiente. |
| --- |

12. **Laziness and concurrency II.** Let us compare the kind of incrementality we get from laziness and from concurrency. Section 4.3.1 gives a producer/consumer example using concurrency. Section 4.5.3 gives the same producer/consumer example using laziness. In both cases, it is possible for the output stream to appear incrementally. What is the difference? What happens if you use both concurrency and laziness in the producer/consumer example?

Resposta:

| Na versão contida na seção 4.3.1, o produtor decide qual o tamanho da stream que será produzida e na seção 4.5.3, o inverso que acontece, o consumidor que irá decidir o tamanho da stream que irá consumir. O comportamento é o mesmo que seja usado paralelismo (threads) e delay ao mesmo tempo. |
| --- |

13. **Laziness and monolithic functions.** Consider the following two definitions of lazy list reversal:

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

What is the difference in behavior between {Reverse1 [a b c]} and {Reverse2 [a b c]}? Do the two definitions calculate the same result? Do they have the same lazy behavior? Explain your answer in each case. Finally, compare the execution efficiency of the two definitions. Which definition would you use in a lazy program?

Resposta:

| Quando a função Reverse1 é chamada, ela apenas irá retornar o valor da função de Rev que irá reverter todos os valores e retornar o resultado ao final. Já a função Reverse2 tem um delay em cada uma de suas operações interativas, tendo apenas um único ‘if’ sendo a seu ponto de saída da recursão de chamadas e que só irá ser chamada ao ser necessário o valor para executar alguma operação ao final, sendo assim mais lento que a implementação do Reverse1. |
| --- |

14. **Laziness and iterative computation.** In the declarative model, one advantage of dataflow variables is that the straightforward definition of Append is iterative. For this exercise, consider the straightforward lazy version of Append without dataflow variables, as defined in Section 4.5.7. Is it iterative? Why or why not?

Resposta:

| Transformando o código para algo mais próximo da linguagem núcleo de Oz iremos encontrar:  declare LAppend  proc {LAppend As Bs ?Rs}  case As  of nil then Bs  [] A|Ar then  local Next in  Rs = A|Next  {LAppend Ar Bs Next}  end  end  end  A cada chamada, a variável Rs guarda os dados, que embora sejam deixados de ser referenciados após a chamada seguinte, acabam sendo recapturados por essa variável que permanece dentro do ambiente. Esse processo se repete de forma apressada e gera o comportamento Lazy esperado na declaração da função. Sendo assim, o comportamento da função não é interativo. |
| --- |

18. **Concurrency and exceptions**. Consider the following control abstraction that implements **try–finally**:

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

Using the abstract machine semantics as a guide, determine the different possible results of the following program:

local U=1 V=2 in (01)

{TryFinally (02)

proc {$} (03)

thread (04)

{TryFinally proc {$} U=V end (05)

proc {$} {Browse bing} end} (06)

end (07)

end (08)

proc {$} {Browse bong} end} (09)

end (10)

How many different results are possible? How many different executions are possible?

Resposta:

| - Quantos resultados são possíveis? 2, sendo a exibição de Bing ou Bong no Browser do ambiente Mozart.  - Quantas diferentes execuções são possíveis? São 6 casos possíveis. |
| --- |

19. **Limitations of declarative concurrency.** Section 4.7 states that declarative concurrency cannot model client/server applications, because the server cannot read commands from more than one client. Yet, the declarative Merge function of Section 4.5.6 reads from three input streams to generate one output stream. How can this be?

Resposta:

| A função Merge mostrada na seção 4.5.6 é uma operação que pega um elemento de cada uma das 3 streams e junta elas, então não há não-determinismo. Cada stream fica esperando sendo limitado pelo consumidor. Não é possível apenas unir duas listas se uma delas não estiver ligada (possuir mais de um elemento). Se você fizer isso, o resultado irá variar dependendo do tempo de união (ordem com os elementos passados pelas streams) e o comportamento não-determinista aparecerá. Os requisitos para aplicativos cliente-servidor não são correspondidos pelo modelo declarativo. |
| --- |