Lista 03 - LIP

Aluno Gabriel Teixeira Queiroz Damasceno - 565118

Questão 1)

Item A)

Todas as possibilidades da execução do statement:

```
thread B=true end
thread B=false end -> error
if B then {Browse yes}

thread B=false end
thread B=true end -> error
if B then {Browse yes}

thread B=true end
if B then {Browse yes} -> yes
thread B=false end -> error

thread B=false end
if B then {Browse yes}
thread B=true end  -> error
```

Item B)

Modificação:

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

Outra possibilidade:

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

Questão 3)

A versão sequencial do Fibonacci é muito mais rápida que a versão concorrente, visto que o número de chamadas recursivas do Fibonacci é de ordem exponencial, sendo que o Fibonacci concorrente cria duas threads a cada chamada. Daí, a quantidade de threads cresce exponencialmente, de forma que o custo para gerenciar as threads passa a ser alto para N grande, podendo até dar erro de excesso de memória devido à criação exponencial de threads:

(N = 35)

```
FATAL: The active memory (732097648) after a GC is over the maximal
heap size threshold: 732096600
Terminated VM 1
```

Além disso, em relação ao tempo, há de se considerar que também há um custo para a criação de cada thread, e esse custo se reflete no tempo: como a quantidade de threads cresce exponencialmente, o tempo de execução do Fibonacci concorrente também explode para N grande.

Questão 4)

Considerando o seguinte código:

```
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}
```

Não importa a ordem em que as threads são criadas, pois o resultado será sempre o mesmo: D = 4. Isso acontece devido ao comportamento dataflow do modelo concorrente declarativo, isto é, caso uma thread tenha uma variável não ligada, ela será suspensa até que essa mesma variável seja ligada, mesmo que em outra thread. Daí, quando houver a ligação, a thread suspensa pode voltar a executar, dando continuidade ao programa. Vale dizer que isso é possível por threads compartilharem o mesmo estado de memória. Assim, trazendo a explicação para nosso exemplo, a linha **`thread D=C+1 end`** só será de fato executada quanto a variável C estiver ligada a algum valor. Caso contrário, a thread será suspensa e

aguardará a ligação. Dito isso, tem-se que a ordem das adições do código concorrente acima é igual à do código sequencial abaixo (adições realizadas de cima para baixo):

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

Questão 5)

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

Reescrevendo em linguagem núcleo:

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

O que podemos perceber é que o statement **`Cond = X == unit`** decide se o programa irá prosseguir ou suspender, por intermédio do comportamento dataflow. Isto ocorre porque X pode ser uma variável não ligada, o que ocasionará uma suspensão no programa até que X receba um valor. Quando isso ocorrer, o procedimento executará a instrução "skip" e dará continuidade a execução do programa: essa definição sintetiza o conceito do Wait, visto que faz com que o programa espere caso uma variável não ligada esteja sendo usada como valor parcial e dá seguimento à execução assim que ela se liga a um valor. Daí, temos que a definição acima é válida.

Questão 8)

Item A)

Ao executar

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

a função Filter será chamada para a lista `[5 1 A 4 0]`, com A sendo uma variável não ligada. Daí, devido ao comportamento dataflow, haverá um momento durante a execução da função tal que `if {F A} then A|{Filter In2 F} else` ..., e o programa será suspenso até que A seja ligada a algum valor. Como isso não ocorre no exemplo do item A, então o procedimento Show não exibirá nada, visto que a função Filter nunca será finalizada.

Item B)

No caso do item B, há a criação de uma thread que executa o filtro e atribui seu resultado a uma variável Out, enquanto, na thread principal do programa, o procedimento Show é chamado. Isso permite um resultado diferente do item anterior, visto que, com a inserção de threads, pode haver resultados parciais, proporcionados pela execução incremental das threads. Assim, a função Filter não precisa agir sobre toda a lista para que o procedimento Show imprima algum resultado. Entretanto, vale dizer que podem haver saídas diferentes, uma vez que o Show exibe o resultado instantâneo, o qual depende da ordem na qual as threads são executadas. Dito isso, segue uma lista com as possibilidades:

_

5|_

Como A ainda é uma variável não ligada, não há como o programa prosseguir. Quanto ao determinismo, é importante frisar: a função Filter continua determinística, uma vez que o estado do programa não muda, independentemente da ordem das threads, e o que muda quanto ao resultado é apenas a saída exibida pelo procedimento Show.

Item C)

Neste caso, devido ao delay, na prática, há tempo suficiente para que a thread criada execute a função Filter até suspender, restringindo o resultado parcial exibido pelo Show a **5|_**. No caso da teoria, ainda assim essa restrição não é garantida, de forma que o resultado _ também é possível.

Item D)

Aqui, a variável A é ligada ao valor numérico 6 dentro de uma thread, e há um delay de 1000ms até a chamada do Show. Daí, na prática, há tempo suficiente para que as duas threads trabalhem para aplicar o Filter à lista, resultando em **5|6|4|nil**. Esse resultado agora é possível pois, mesmo que a execução de Filter suspenda, logo a thread correspondente voltará a executar, visto que a variável A foi ligada a um valor dentro de outra thread, que, devido ao delay, será executada antes do procedimento Show. Entretanto, na teoria, tal como no caso do item C, não há garantia de que o delay será suficiente para restringir a saída exibida, de forma que as seguintes possibilidades também são possíveis:

_
5|_
5|6|_
5|6|4|_
5|6|4|nil

Questão 10)

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

Calcular {Three} + 0 retorna 3 depois de 1000ms. O que acontece ao calcular {Three}+0 três vezes?

```
{Three}+0
{Three}+0
{Three}+0
```

Reescrevendo

```
local A0 B0 C0 A1 B1 C1 Z
     Z = 0
     A0 = {Three}
     A1 = A0+Z

     B0 = {Three}
     B1 = B0+Z

     C0 = {Three}
     C1 = C0+Z
end
```

Assim, podemos ver com mais clareza que a função é executada três vezes, visto que o valor retornado (3) não é guardado em uma variável para futura utilização. O que acontece é que a função é chamada três vezes, de forma que, a cada cálculo, o gatilho para a função lazy Three é ativado e 3 é retornado. Daí, a cada cálculo a função leva 1 segundo para retornar o resultado 3, o que resulta um tempo total de 3 segundos para a execução do programa.

Questão 11)

Reescrevamos um trecho do código utilizando a linguagem núcleo

```
local A0 A1 in
     A0 = X + Y
     A1 = A0 + Z
     {Browse A1}
end
```

Aqui, vemos que em **`A0 = X + Y`** há um gatilho para a execução das funções lazy MakeX e MakeY, que retornam valores para as duas variáveis a serem somadas. Daí, como a soma demanda pelos argumentos de forma concorrente, visto que os cálculos de funções lazy são executados em threads, os átomos x e y são exibidos simultaneamente. Após isso, X se liga ao valor 1 após 3 segundos, enquanto Y se liga a 2 após 6 segundos, e finalmente a soma é realizada, resultando em A0 = 3. Agora, a soma **`A0 + Z`** será realizada, a qual ativa um gatilho para a execução da função MakeZ, que exibe o átomo z (6 segundos após exibir x e y, uma vez que levou 6 segundos para que A0 recebesse o valor da soma). Assim, após 9 segundos, a função retorna 3 como valor de Z e A1 recebe o valor 6. Por fim, A1 = 6 é exibido ao final do programa, o qual durou 15 segundos.

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

```
local A0 A1 in
     A0 = Y + Z
     A1 = X + A0
     {Browse A1}
end
```

Neste caso, A0 ativa o gatilho para MakeY e MakeZ, exibindo y e z simultaneamente, e retornando os argumentos da soma 9 segundos depois, cujo resultado é *A0 = 5*. Após isso, em A1 há o gatilho para MakeX, que exibe o átomo x e retorna o valor *X = 1* 3 segundos depois, finalizando assim a soma e exibindo A1. Daí, o programa durou 12 segundos.

```
local A0 A1 in
     A0 = thread X+Y end
     A1 = A0 + Z
     {Browse A1}
end
```

Já aqui, como há a criação de uma thread, há duas possibilidades para a ordem de execução:

```
A0 = thread X+Y end
A1 = A0 + Z
```

ou

```
A1 = A0 + Z
A0 = thread X+Y end
```

Os dois casos são equivalentes em termos de tempo, visto que, devido a thread, (X + Y) e Z ativam os gatilhos simultaneamente. Daí, a operação X + Y é calculada em 6 segundos (X + Y = 3), enquanto Z recebe 3 em 9 segundos. Daí, após esses 9 segundos, a operação A0 + Z finalmente pode ser executada, resultando em A1 = 6. Daí, o tempo de execução do programa é de 9 segundos.

Por fim,, tem-se que o mais rápido é o programa que calcula *thread X+Y end + Z.*

How would you program the addition of n integers i1,..., in, given that integer ij only appears after tj ms, so that the final result appears the quickest?

Pela lógica vista até o momento, a melhor ideia é agrupar entre parênteses os pares de variáveis cujos tempos de retorno são maiores, calculando cada soma por intermédio de thread. Assim, apenas o maior tempo tj será relevante, evitando somas de tempo. Daí, quanto menor o tempo tj, mais externa estará a variável na cadeia de parênteses, da seguinte maneira:

i1 + thread (i2 + thread (i3 + (...))), com tj de i1 < tj de i2, tj de i2 > tj de i3 e assim por diante, para que os gatilhos das variáveis com maior delay tj sejam ativados primeiro.

Questão 13)

```
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
```

Suponha o seguinte trecho:

```
L = {Reverse1 S}
{Browse L.1}
```

O que acontece é que L.1 serve de gatilho para a execução do Reverse1, que internamente chama e executa a função Rev para inverter a lista S. Entretanto, Rev é uma função monolítica, de forma que se faz necessário que ela execute até o caso base (isto é, percorra a lista S inteira) para ser capaz de retornar o primeiro elemento, visto que o primeiro elemento

só é calculado no final. Daí, não temos a característica da execução incremental para este caso.

Suponha agora

```
L = {Reverse2 S}
{Browse L.1}
```

O L.1 é responsável por ativar o gatilho para a função lazy Reverse2, que, tal como Reverse1, chama e executa a função Rev, que, desta vez, é uma função Lazy. Entretanto, ainda assim, para retornar a primeira posição da lista invertida, é preciso que Rev inverta a lista inteira, independentemente de ser uma função lazy. Dito isso, o que muda em relação a Reverse1 é que, por Rev ser lazy, há a ativação de muitos gatilhos desnecessários (se N é o tamanho da pilha, então há N gatilhos ativados), visto que Rev é monolítica.

Em resumo, ambas não só exibem o mesmo resultado como também possuem o mesmo comportamento (calculam o resultado de maneira monolítica, sem execução incremental). Entretanto, o Reverse2 acaba sendo menos eficiente do que o Reverse1, justamente por ativar muitos gatilhos sem necessidade.

Questão 14)

```
fun lazy {LAppend As Bs}
     case As of nil then Bs
     [] A|Ar then A|{LAppend Ar Bs} end
end
```

Reescrevamos em linguagem núcleo:

```
local P in
   proc {P As Bs R}
      case As of nil then
         R = Bs
      [] A|Ar then
         local S in
            {ByNeed {LAppend Ar Bs R} S}
            R = A|S
         end
```

```
      end
   end
```

Como vemos, por não haver variável dataflow, há uma operação a ser realizada após a chamada recursiva, o que, em uma situação normal, causaria acúmulo de operações na pilha de recursão. Entretanto, como LAppend é uma função lazy, o trigger ByNeed vai executar somente quando houver gatilho. Daí, é como se as chamadas recursivas não fossem executadas imediatamente, e sim encapsuladas até o momento em que um certo elemento da lista resultante do LAppend é necessário. Assim, `R = A|S` é inicialmente executado antes da chamada recursiva. Logo, pode-se caracterizar a função LAppend como iterativa mesmo sem otimização da chamada pela cauda proporcionada pelo comportamento dataflow, visto que também não há acúmulo de operações na pilha.

Questão 18)

```
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
```

O que podemos ver, olhando de dentro para fora, é que o procedimento S1 do TryFinally mais interno vai ocasionar uma exceção, visto que U aponta para um valor diferente de V. Entretanto, S2 sempre é executado, o que exibe “bing”. Daí, já sabemos que o programa sempre executará {Browse bing}. Dito isso, considerando o caso em que a thread criada é escolhida primeiro pelo escalonamento, o programa exibe o bing, a thread finaliza, o procedimento S2 do TryFinally externo é executado e o programa exibe bong. Além disso, outra possibilidade é, no TryFinally externo, o procedimento S2 executar primeiro, resultando na exibição de bong antes de bing. Logo, em resumo, temos dois resultados possíveis: *bing bong* e *bong bing*.