Lista 00 - LIP

Aluno Gabriel Teixeira Queiroz Damasceno - 565118

Questões 1 e 2 estão no código.

Questão 3) Usemos indução para provar a corretude da função {Pascal N}.

Objetivo: calcular a N-ésima fileira do triângulo de pascal.

Base: N = 1. Assim, o programa entra na instrução "if", que retorna uma lista com o elemento 1 ([1]), o que está correto, visto que a primeira fileira do triângulo de pascal contém apenas 1.

Passo: suponha, para N>1, que {Pascal N-1} está correto, isto é, que a fileira N-1 do triângulo de pascal foi calculada corretamente. Mostremos que {Pascal N} também está correta.

Assim, como N>1, o programa executa a instrução “else”, que corresponde à linha *{AddList {ShiftLeft {Pascal N-1} {ShiftRight {Pascal N-1}}}.* Como provado abaixo, as funções ShiftLeft, ShiftRight e AddList estão corretas, logo, nos resta a chamada {Pascal N-1}, que, pela hipótese indutiva, também é válida. Portanto, o algoritmo criará duas listas contendo a fileira N-1, porém uma (digamos Le) com um 0 adicionado ao início (resultado de ShiftRight) e outra (digamos Ld) com um 0 adicionado ao final (resultado de ShiftLeft). Assim, temos

Le = [0 1 X2 ... XN 1]

Ld = [1 X2 ... XN 1 0]

Depois, somará as duas, elemento a elemento, por meio da função AddList, resultando na suposta N-ésima fileira do triângulo de pascal, digamos Ls.

Ls = [1 (1 + X2) (X2 + ... + 1) 1]

Nota-se que há o número 1 em ambas as bordas, como em toda fileira do triângulo de pascal, e que todo elemento presente na fileira é calculado como a soma de dois elementos adjacentes da fileira anterior (se não há elemento adjacente, consideramos 0): {0 +1, 1+X2, X2 + X3, ..., XN + 1, 1 + 0}.

Como a fileira N-1 está correta, e o algoritmo para se calcular a N-ésima fileira do triângulo de pascal é respeitado, a fileira N também está correta.

Corretude das funções ShiftLeft, ShiftRight e AddList:

Corretude {ShiftLeft L}:

Objetivo: adicionar um 0 no final da lista L.

Base: L = nil, isto é, lista L vazia. Neste caso, o programa entra na instrução "else" e retorna [0]. Como a lista estava vazia, então o programa adicionou 0 no final, como desejado.

Passo: suponha que, para uma dada lista T não vazia, digamos [X1 X2 ... Xn], o programa está correto, isto é, {ShiftLeft T} adiciona um 0 no final da lista T. Mostremos que para L = H|T ele também está.

Ao chamar {ShiftLeft L}, o programa entra na instrução "case", que realiza o *pattern matching*, desmembrando a lista L em H|{ShiftLeft T}. Pela hipótese indutiva, {ShiftLeft T} vale, o que nos dará a lista [X1 X2 ... Xn 0]. Daí, a lista H|T, ao final do programa, corresponde à lista L (cuja cabeça é H e a cauda é T) com o inteiro 0 adicionado ao final, como desejado.

Corretude {ShiftRight L}:

Objetivo: adicionar 0 no início da lista L.

A função ShiftRight possui a instrução 0|L, que, independentemente de L, retorna uma lista que tem como cabeça o elemento 0 e como cauda a lista L, como desejado.

Corretude {AddList L1 L2}:

Objetivo: somar cada elemento de L1 ao elemento de L2 de posição correspondente.

Hipótese: considerando a aplicação do AddList no contexto da função Pascal, L1 e L2 serão listas com a mesma quantidade de elementos.

Base: listas L1 e L2 iguais a nil, isto é, vazias. Neste caso, o programa entrará na instrução "else" e retornará nil. Por vacuidade, a propriedade vale.

Passo: suponha que o programa vale para T1 e T2, com n > 0 elementos, isto é, que o programa soma os elementos de T1 com os elementos de posição

correspondente de T2. Mostremos que ele vale para L1 = H1|T1 e L2 = H2|T2. Daí, como L1 e L2 não são vazias, ambas as instruções case serão válidas e o programa executará “H1+H2 | {AddList T1 T2}”, que somará as cabeças H1 e H2 e chamará {AddList T1 T2}, que, pela hipótese indutiva, é válida. Assim, o programa retornará uma lista cuja cabeça é a soma das cabeças de L1 e L2 e cuja cauda é a soma dos elementos das caudas de L1 e L2, resultando na soma completa das listas, como desejado.

Questão 4)

A seção 1.7 compara a versão exponencial do Pascal com a versão quadrática. Daí, entendemos que programas com complexidades exponenciais são inviáveis, visto que aumentam em uma taxa muito alta conforme a entrada cresce, o que os torna lentos facilmente. Já no caso dos algoritmos de complexidade polinomial de baixa ordem, por mais que tenhamos, a exemplo, uma complexidade de ordem quadrática, que em geral não é considerada muito eficiente, ainda assim eles são mais praticáveis que os algoritmos exponenciais e podem ser úteis em muitos casos. No caso de algoritmos polinomiais de alta ordem, geralmente eles não são considerados eficientes, dado que a ordem do polinômio é muito alta, o que os faz explodir com o aumento da entrada, tornando-os impraticáveis.

Questão 5)

No caso de chamarmos {SumList {Ints 0}}, após um tempo calculando o resultado, teoricamente infinito, o programa apresentará um erro de excesso de memória:

```
FATAL: The active memory (732096642) after a GC is over the maximal heap size threshold: 732096600.
```

Isso acontece porque a função Ints calcula uma lista infinita ao ser chamada por SumList, que tenta somar todos os valores dessa lista, o que, além de não ser uma boa ideia, não funciona.

Questão 6) – complemento

Utilizando a operação de multiplicação, obtemos um triângulo de pascal apenas de zeros, exceto pela primeira fileira, que é o caso base [1]. Isso acontece porque, logo na segunda linha, temos de multiplicar 1 por 0 para gerar seus dois

elementos , visto que não há nenhum outro número adjacente ao 1 na primeira fileira. Daí todas as multiplicações seguintes passam a ter o 0.

Já no caso da operação Mul1, definida no enunciado da questão, o resultado da décima fileira é o seguinte:

```
[10 6235300 160344312228246705825060
 491169465008447673987143401842548715992798136448 44
 505756353132539355991535788904396967700352784465846135098 29
 505756353132539355991535788904396967700352784465846135098 29
 491169465008447673987143401842548715992798136448 44
 160344312228246705825060 6235300 10]
```

Agora, o 0 que representava o número adjacente às extremidades é acrescido em 1, o que tira o 0 das multiplicações.

Questão 7)

No primeiro fragmento de código, há dois escopos: o primeiro e mais externo, onde X recebe o valor 23, e o segundo e mais interno, onde X recebe 44. Como o Browse encontra-se no escopo local mais externo, isto é, onde 23 foi atribuído a X, então {Browse X} exibe 23. Já no segunto caso, X é uma célula, cujo valor pode ser alterado. Daí, em um mesmo escopo, X é inicializado com o valor 23 e logo é alterado para o valor 44. Assim, o Browse exibe o valor 44.

Questão 8) - complemento

O problema dessa definição está relacionado ao escopo da variável Acc, isso porque, na linha “Acc in”, o programa delimita o escopo da variável, que se torna visível somente dentro da função. Para contornar esse problema, basta tornar Acc uma variável global, declarando-a fora da função Accumulate, ou criar um escopo maior que englobe tanto o Accumulate quanto o Browse, lembrando-se de criar a célula fora da função Accumulate.

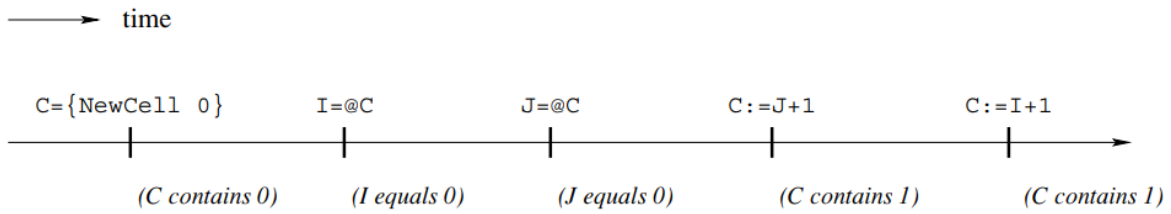
```
declare Acc
Acc={NewCell 0}

declare
fun {Accumulate N}
    Acc:=@Acc+N
    @Acc
end

{Browse {Accumulate 5}}
{Browse {Accumulate 100}}
{Browse {Accumulate 45}}
```

Questão 9) está no código.

Questão 10)

Item a)

Rodei o programa 100 vezes, mas infelizmente não obtive 1 como resultado em nenhuma delas. Entretanto, embora seja atípico, é possível que isso aconteça devido ao não determinismo, isto é, por não haver exatidão quanto à ordem em que cada instrução será executada, que pode ser diferente até para programas iguais. Daí, pode ocorrer a chamada interleaving, ou intercalação, visto que as threads são executadas de maneira intercalada, e não de uma só vez. Assim, existem diferentes configurações de intercalação, devido ao não determinismo, que podem ocasionar resultados diferentes em um programa que alia estado explícito com concorrência. No caso no programa da seção 1.15, existe a seguinte possibilidade:



I recebe C, que é 0, e logo depois J também recebe C, ainda com o valor 0, visto que as instruções que incrementam C ainda não foram executadas. Assim, C receberá, ao final do programa, o valor 0 + 1 duas vezes, resultando em 1.

Isto é um claro exemplo de race condition, isto é, de quando o não determinismo se torna observável.

Item B) - está no arquivo .oz

Item C)

Utilizando atomicidade para resolver o problema da interleaving, o programa cria blocos de código que envolvem os comandos de cada thread por meio de um lock, cuja função é "trancar" as instruções de uma thread enquanto a outra executa, sendo que a thread trancada só irá executar após o término da outra. No nosso caso, um lock L foi criado na linha `L={NewLock},` envolvendo as instruções de cada thread em uma espécie de "região" que executa os comandos de forma sequencial até o programa encontrar um "end".

Código:

```
declare
C={NewCell 0}
L={NewLock}
thread
   lock L then I in
      I=@C
      C:=I+1
   end
end
thread
   lock L then J in
      J=@C
      C:=J+1
   end
end
```

Só após isso, o programa partirá para a execução da próxima thread, assim impedindo os diferentes resultados causados pela intercalação das threads. Portanto, utilizar delay iria somente atrasar o programa, não mudando em nada o resultado final, que, neste caso, será sempre 2.