Trabalho Prático 2 - PCF

Jéssica Fernandes (A93318) Vitor Lelis (PG54273)

April 07, 2024

Contents

1. Introdução

No decorrer de uma das aulas de Programação Cíber-Física, foi-nos apresentado o problema dos Aventureiros. Este problema consiste na travessia de 4 pessoas de uma margem de uma ponte para a outra, com apenas 1 lanterna, que deve acompanhar o grupo sempre durante as suas travessias. Apenas 2 pessoas podem estar na ponte em simultâneo. O objetivo é saber se este trajeto pode ser efetuado pelas 4 pessoas em 17 minutos ou menos, sabendo que cada uma delas, leva um tempo diferente para atravessar.

Este documento tem como objetivo responder às perguntas do Trabalho Prático 2, que tem como contexto o problema dos Aventureiros acima descrito.

Em primeiro lugar, devemos preencher as funções inacabadas de um ficheiro base em Haskell onde se apresenta modelado o problema. De seguida, devemos comparar esta solução com a realizada na linguagem UPPAAL, em contexto de aula, aquando da apresentação do problema. O objetivo será aplicar os conceitos Haskell aprendidos para resolver o problema nessa linguagem, e posteriormente comparar as duas abordagens estudadas, apontando as vantagens e desvantagens de ambas na resolução deste problema.

1.1. Restrições do Problema:

Como mencionado, começamos a modelação com uma base já feita, mas recapitulemos as restrições do problema:

1. Apenas 2 pessoas, no máximo, em simultâneo, em cima da ponte;
2. Apenas 1 lanterna que deve ser transportada em todas as travessias feitas;
3. O tempo que cada um dos 4 aventureiros demora a atravessar: 1, 2, 5 e 10 minutos, respetivamente.

O objetivo final é que todos os aventureiros se encontrem acompanhados pela lanterna, no lado direito da ponte.

O que queremos testar é se este objetivo final pode ser alcançado em 2 condições:

1. em até 17 minutos, em até 5 travessias;
2. em menos de 17 minutos (neste caso, a intenção é provar a impossibilidade).

2. Modelação do Problema e Verificação da Solução em Haskell

Tendo em conta as condições descritas acima em mente, partimos para a implementação das funções desejadas de forma a provar os pontos propostos.

A primeira coisa a fazer é definir como serão feitas as travessias. Devido à condição da travessia só poder ser efetuada, com a lanterna, definimos que para atravessar para o lado correto (o direito), atravessam 2 aventureiros com a lanterna e, para permitir a travessia dos que continuam no lado esquerdo, apenas **1 aventureiro** (com a intenção de poupar tempo) deve voltar para o lado esquerdo trazendo a lanterna.

2.1. Funções

2.1.1. `getTimeAdv`

A implementação da primeira função foi simples, pois havia poucos casos de aventureiros. Apenas abordamos cada um deles e retornamos o valor correto.

2.1.2. `allValidPlays`

Esta função gera todas as possíveis jogadas a partir de um estado fornecido. A estratégia utilizada foi, primeiro, conferir em qual lado está a lanterna e, depois, criar uma lista de pares com os possíveis aventureiros que podem realizar a travessia.

No entanto, foi estabelecido que apenas 1 aventureiro volta para o lado esquerdo trazendo a lanterna. Neste caso, de forma a evitar um excesso de funções para casos isolados, optou-se por criar uma lista com os pares com o mesmo aventureiro, aproveitando as funções já feitas feitas e aplicando corretamente a lógica.

Para realizar esta operação, foram criadas três funções auxiliares que isolavam as tarefas e tornavam o código mais limpo e coeso:

1. `validPairs` : Cria os tipos de pares necessários para a travessia;
2. `makeMove` : Realiza os movimentos para cada um dos pares;
3. `stateChange` : Troca o estado do par e da lanterna dependendo de seu cenário.

2.1.3. `exec`

Para a implementação do `exec` , foi proposta a criação de uma função recursiva que executaria `n` vezes a `allValidPlays` , acumulando assim os estados possíveis de se jogar e verificando as condições do estado quando chegamos ao estado final. Nesta função, aproveitou-se das características do monad do `ListDur` para obter a maior eficiência possível e coesão do código.

2.1.4. `leq17` e `l17`

Para as funções de verificação, optou-se por abordar estratégias quase idênticas, tendo atenção apenas à condição de sucesso.

No caso da `leq17` , estabeleceu-se que o tempo da `Duration` seria menor ou igual a 17 minutos, e que o máximo de movimentos possíveis era 5 (fornecendo 5 como argumento à função `exec`), comprovando a propriedade de que é possível completar o desafio nesse tempo com essa quantidade de movimentos.

Para o `l17` , a condição da `Duration` mudou para ser menor que 17 minutos, mantendo-se o número de movimentos em 5. Como foi estabelecida a regra de que 2 aventureiros devem atravessar e 1 voltar, no cenário de 4 aventureiros, 5 seria a quantidade máxima de jogadas. Assim, manteve-se esse valor e provou-se que, de facto, não é possível completar o desafio em menos de 17 minutos, qualquer que seja a sequência de jogadas.

3. Comparação entre Haskell e UPPAAL

A solução em Haskell define um conjunto de tipos e funções para modelar o problema através do uso de monads de duração. A sua solução avalia todas as possíveis sequências de movimentos, e verifica se é possível atravessar a ponte em menos de 17 minutos.

Esta abordagem é funcional, assim como a própria linguagem, o que pode ser mais intuitivo na modelação de estados e transições. Além disso, o código é flexível pois está dividido em módulos, o que permite facilmente a modificação, reutilização e extensão para diversos cenários.

Por outro lado, as verificações são feitas de forma manual, com testes, porque Haskell não fornece verificação formal, ao contrário de UPPAAL, que tem esta funcionalidade implementada. Além disto, pode ser um pouco confusa a implementação devido à complexidade dos conceitos de monads e dos tipos utilizados. Em Haskell, por norma, a maior dificuldade é a tipagem, que se torna cada vez mais confusa e difícil de trabalhar quanto mais ambicioso for o problema e, por sua vez, a implementação.

UPPAAL, por sua vez, utiliza autómatos de tempo. Fornece uma verificação formal que garante a correção em termos de tempo e estados, o que, neste contexto, é precisamente o que procuramos. A modelação temporal é precisa e natural pois o programa foi criado para lidar com esse tipo de sistemas, o que o torna indicado para resolução de problemas como o que pretendemos resolver.

Além disto, apresenta uma particularidade que consideramos ser muito importante, e uma das principais em relação ao Haskell que é o facto de não ser uma linguagem escrita. Os modelos e resultados são facilmente visualizados e por isso também mais intuitivos até certo nível. O programa permite fazer simulações e análises detalhadas. Consideramos ser uma grande vantagem o facto de podermos visualizar o sistema pois a compreensão é facilitada.

No entanto, a criação de modelos em UPPAL também pode tornar-se complexa muito facilmente, principalmente quanto maior e mais complexo for o sistema que se pretende representar.

Ainda assim, apesar de consideramos que UPPAAL pode ser difícil de utilizar para modelação, principalmente quando ainda nos estamos a familiarizar com a ferramenta, concluímos que, depois de prontos, os modelos nesta linguagem são mais legíveis e compreensíveis em relação ao Haskell. Isto deve-se principalmente ao facto de serem visuais como já mencionado.

4. Conclusão

Neste relatório, abordamos a modelação e análise de duas abordagens distintas - Haskell e UPPPAL, de resolução do problema já conhecido dos Aventureiros.

Em Haskell, utilizamos conceitos de monads para modelar o problema e verificar se o desafio poderia ser finalizado em até 17 minutos. Demonstramos que, com a estratégia correta, é possível atingir o objetivo no tempo proposto e provamos que é impossível completá-lo em menos de 17 minutos. Fizémo-lo calculando todas as possíveis sequências de estados a partir do inicial, e verificando sob quais condições se encontravam os vários estados finais possíveis. A abordagem funcional de Haskell é poderosa e flexível, mas complexa devido à tipagem e aos conceitos avançados de programação.

UPPAAL, utilizando autómatos temporais, forneceu uma verificação formal e precisa, com simulações e análises detalhadas. A visualização intuitiva dos modelos facilita a compreensão e a verificação dos estados e transições, apesar da possível complexidade na criação dos modelos.

Comparando as duas abordagens, concluímos que Haskell oferece flexibilidade e modularidade, ideal para problemas complexos, enquanto UPPAAL proporciona uma verificação formal robusta e uma visualização intuitiva. Cada uma das duas ferramentas apresenta pontos fortes e fracos, e a escolha entre Haskell e UPPAAL dependerá do que o programador mais valorizar em termos de conforto da ferramenta. Porém, a nossa conclusão é que apesar de Haskell ser uma linguagem mais familiar, UPPAAL é mais intuitivo e compreensível.

Este projeto serviu não só o propósito de consolidar os conceitos monádicos aprendidos com um exemplo prático, mas também de obter duas resoluções do mesmo problema em linguagens diferentes, e poder comparar o desempenho de ambas exatamente nas mesmas condições, tirando conclusões relevantes sobre o uso de cada uma.