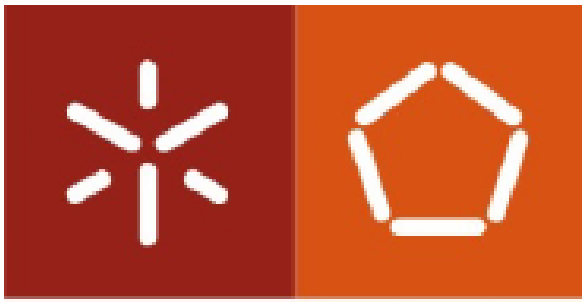

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Arquitectura e Cálculo

Adventures in a Rope-Bridge

Lucas Pereira A68547

Ricardo Pereira A73577

June 16, 2020

1 Introdução

Este projeto consistia na resolução de um problema em que têm de ser tomadas decisões dependentes do tempo, tal como no projeto anterior, com a diferença de que esta utilizará outra ferramenta para fazer essas tomadas de decisão. Os dados do problema são exatamente os mesmos, envolvendo assim quatro aventureiros e uma ponte que os mesmos têm que atravessar sempre acompanhados de uma lanterna, contudo existem algumas restrições que os mesmos têm que cumprir:

- a ponte deve ser atravessada no máximo por dois aventureiros;
- apenas existe uma lanterna;
- cada aventureiro demora um período diferente a atravessar a ponte (um, dois, cinco e dez minutos).
- quando dois aventureiros atravessam a ponte, o tempo dessa travessia é igual ao tempo de travessia do aventureiro mais lento;

Tendo em conta estes factos e condições, pretende-se saber qual o tempo total da travessia feita pelos quatro aventureiros. A solução deve ser dezassete minutos e menos do que isso deve ser impossível, algo que deve ser comprovado desta vez com a ajuda do *Haskell*.

2 Resolução

Posto isto, no ficheiro providenciado pelo docente existiam várias funções por completar, sendo também necessário definir o comportamento da estrutura monádica. Foram ainda definidas outras funções que de alguma forma ajudaram na resolução do problema tendo-se primado por não definir qualquer função já existente em bibliotecas. Assim, começou-se pela contrução da estrutura monádica.

2.1 Estrutura Monádica

2.1.1 Definição do Tipo de Dados

O tipo de dados *ListDur a* é composto pelo construtor *LD* ao qual se sucede uma lista de elementos do tipo *Duration a*. Por sua vez, este último é composto pelo contrutor *Duration* ao qual sucede um par cujo primeiro elemento é um inteiro (que há de ser o tempo) e o segundo, um elemento de um tipo qualquer. Importa desde já dizer que o tipo deste segundo elemento do par irá ser uma lista de *Booleans* (que representa as posições de cada interveniente no problema) como veremos mais à frente. Informalmente, pode-se dizer que a estrutura de dados é uma lista de pares em que o segundo elemento de cada par é a lista de posições dos aventureiros e da lanterna e o primeiro o tempo decorrido até atingirem essas posições. Está assim explicada a estrutura de dados cuja a compreensão é fundamental.

2.1.2 Definição do Funtor

A definição do funtor permite definir o *map* aplicado ao tipo de dados *ListDur a*. Posto isto, sempre que se quiser aplicar uma função a cada elemento de um *ListDur a*, utiliza-se a função *fmap*.

2.1.3 Definição do Aplicativo

A definição do aplicativo permite dizer aquilo que acontece quando se executa a função $(<*>)$ aplicada a dois *ListDur a* sendo que um deles vem encapsulado com funções que serão aplicadas aos elementos do outro *ListDur a* que estão nas posições homónimas. Por exemplo, (LD [Duration(2,(+1)), Duration(3,(*10)), Duration(4,(*200))]) <*>(LD [Duration(0,3), Duration(2,8), Duration(1,1)]) dará como resultado LD [Duration (2,4),Duration (3,80),Duration (4,200)] .

2.1.4 Definição da Monada

A definição desta monada permite trabalhar com a estrutura de dados de uma forma muito mais simples, uma vez que os elementos estão todos envoltos em contextos. Outra vantagem associada à definição da monada é a de permitir usar o *monadic binding*.

Estando a estrutura monádica a funcionar devidamente, partiu-se para a definição das funções enunciadas a seguir. Algumas já existiam outras sofreram pequenas alterações uma vez que da forma como estavam definidas não permitiam resolver o problema.

2.2 Definição das funções

- ***allValidPlays*** - Dado um *State* era necessário calcular todas as variações possíveis do mesmo, e para servir esse propósito existe esta função. Assim, o cálculo de uma variação consiste na passagem de uma lista de *Objects* e do estado a modificar à função *mChangeSate*, fazendo-se ainda posteriormente a adição do tempo da travessia depois de se ter o valor de retorno do processo anterior;
- ***exec*** - A execução do programa está principalmente assente nesta função que dado um estado inicial e um número travessias retorna todos os estados e respetivas durações que foram registadas. Aqui, a definição da monada é extremamente útil visto que permite a utilização do *monadic binding*. Pode-se ainda verificar que existe recursividade, solução encontrada para que o número de travessias possíveis fosse respeitado;
- ***getTimeAdv*** - Esta função retorna o tempo de um aventureiro, dado esse mesmo aventureiro;
- ***gFinal*** - A contrução do estado final não era estritamente necessária mas permite fazer a abstração da linguagem de programação;
- ***leq17*** - Serve esta função para garantir que é possível encontrar um estado em que os aventureiros (e a lanterna) estão todos do lado de lá da ponte e conseguiram atravessá-la em menos que ou igual a 17 unidades temporais;
- ***l17*** - Praticamente igual à anterior, a função *l17* verifica se a travessia pode ser feita em menos que 17 unidades temporais;
- ***mChangeState*** - Apesar de estar previamente definida, esta função necessitou de uma alteração para que o estado apenas fosse modificado se os *Objects* recebidos estivessem todos no mesmo lado da ponte (valores boleanos iguais no *State*). Caso não estejam, o estado permanece igual;

2.3 Resultados

Executando o programa é possível ver que o mesmo corre da forma que é suposto, sendo que as funções ***leq17*** e ***l17*** garantem que os resultados são exatamente os mesmos que no projeto anterior onde se utilizou outra ferramenta - o *UPPAAL* - sendo o valor temporal mínimo possível da travessia de 17 unidades de tempo após 5 travessias.

2.4 *Haskell VS UPPAAL*

Tendo-se agora uma prespetiva das duas ferramentas na resolução de problemas deste género, é possível compará-las e tirar algumas conclusões relativamente aos prós e aos contras de ambas:

- O **aspeto visual** tem, como sabemos, um grande impacto na performance do ser humano compreender os problemas, sendo assim o *UPPAAL* melhor que o *Haskell* visto que se consegue **visualizar o autómato** com todas as condições e consequências de cada ação, funcionando quase como um esquema que é compreensível até por quem não tem experiência no assunto. Imagine-se que existe a **necessidade de explicar o problema a alguém externo à área** em questão, como se faria em *Haskell*? Provavelmente, **seria necessário explicar também o funcionamento da linguagem** e de cada função;
- O *UPPAAL* permite a utilização de **lógica temporal**, algo que é bastante útil para verificar determinadas propriedades temporais sobre um problema, contudo, após alguma pesquisa, reparou-se que **também existem bibliotecas em *Haskell*** que o permitem fazer, e não existindo, **seria sempre possível contruir funções para esse efeito**, o que leva ao próximo ponto;
- O que se verifica no ponto anterior é uma outra característica muito vantajosa do *Haskell* - existe uma **maior liberdade para a implementação de problemas em *Haskell*** do que em *UPPAAL*, visto que neste último há a **obrigatoriedade de seguir a metodologia dos autómatos**;

- A resolução de problemas em *Haskell* utiliza (obviamente) uma linguagem de programação, empregando assim o paradigma declarativo da programação funcional que é **mais próximo daquilo que a grande generalidade dos programadores estão habituados** a fazer (embora trabalhem mais com outros paradigmas). Contudo, é necessário conhecer determinados conceitos da linguagem, assim como em *UPPAAL* é necessário conhecer a ferramenta;
- A execução do programa em *Haskell* **é mais transparente** na medida em que se consegue ver tudo aquilo que está a acontecer (conteúdo de variáveis, tipos, etc), já no *UPPAAL* o utilizador está **limitado àquilo que o mesmo oferece**.

Tendo em conta os aspetos anteriores, achamos que **o *Haskell* sobressai em termos de flexibilidade** comparativamente a clareza visual que o *UPPAAL* é capaz de oferecer.

3 Conclusão

O problema dos aventureiros permitiu tanto no projeto anterior como no atual, atestar a eficácia desta ferramenta para problemas do mesmo género, onde devem ser tomadas decisões que se querem o mais otimizadas possível. Desta vez, foi possível explorar outras áreas que esta linguagem pode abranger, verificando-se assim a utilidade da programação nos mais simples problemas que podemos encontrar no nosso quotidiano.

Fica ainda por concluir a implementação da impressão do caminho final dos aventureiros que deverá ainda ser apresentada na defesa do projeto.