

Arquitectura & Cálculo

Trabalho Prático 3

MiEI — Ano Lectivo de 2016/17

Departamento de Informática
Universidade do Minho

A70430 João Bernardo Machado Quintas Dias da Costa
A72205 Luís Martinho de Aragão Rego da Silva
A71580 Rafael Alexandre Antunes Barbosa

Junho de 2017

1 Dependências

Inicialmente são feitos os *imports* das bibliotecas fornecidas na página da disciplina, necessárias para desenvolver o trabalho prático.

```
import Cp
import SMT
import MMM
import Qais
import Probability
```

O grupo utilizou também algumas das funções desenvolvidas no decorrer das aulas práticas, nomeadamente as funções que definem o funcionamento de uma *stack* (baseada em listas).

```
push = flip (:)
pop = tail
top = head
empty = (0  $\equiv$ ) · length
```

Também as respectivas “*totalizações*” das funções apresentadas acima foram adoptadas neste projeto, pois irão ser necessárias mais à frente.

```
top' = (tot (split id top) ( $\neg$  · empty)) ·  $\pi_1$ 
pop' = tot (split pop top) ( $\neg$  · empty) ·  $\pi_1$ 
push' = return · (split  $\widehat{\text{push}}$  bang)
```

2 Caso de Estudo

O caso de estudo consiste na implementação de uma *Queue* composta por duas *Stacks*:

- Stack de *dequeue* - *stack* da esquerda no *State* definido pelo grupo;
- Stack de *enqueue* - *stack* da direita no *State* definido pelo grupo.

Com esta topologia do estado da *Mealy Machine* pretendida, procedemos à definição base dos métodos de uma *Queue* : *Enqueue*, *Dequeue* e *Peek*.

$$\begin{aligned} enq &= extl \text{ push}' \\ deq &= extr \text{ pop}' \\ peek &= extr \text{ top}' \end{aligned}$$

Para permitir o comportamento de *flush* dos elementos da *stack* de *enqueues* para a *stack* de *dequeues*, desenvolveu-se a função seguinte:

$$flush = split ((\widehat{foldl \text{ push}}) \cdot swap \cdot (reverse \times reverse)) []$$

De realçar que esta função faz um *flush* naïve, i.e., coloca sempre os elementos da *stack* de *enqueues* na *stack* de *dequeues* através de *pops* e *pushs* sucessivos.

Posteriormente, partiu-se para a extensão destas funções para terem o comportamento desejado. Tanto a função *enqueue*, como a função *peek*, não sofreram quaisquer alterações pois já tinham o comportamento desejado.

No entanto, a função *flush* teve de ser extendida para apenas realizar a acção de *flushing* quando a *stack* de *dequeues* estiver vazia. Desta forma, o *flush* não é realizado desnecessariamente quando a *stack* em questão tem elementos que podem ser retirados.

$$\begin{aligned} enq' &= enq \\ peek' &= peek \\ flush' &= return \cdot split (Cp.cond (empty \cdot \pi_1) flush id) bang \cdot \pi_1 \end{aligned}$$

A função *dequeue* também teve de ser extendida para representar o comportamento desejado. A nova função desenvolvida, *deq'*, retorna *Nothing* caso não haja elementos em nenhuma das *stacks*, como é pretendido, faz *flush* na situação em que tem elementos apenas na *stack* de *enqueues*, e faz *enqueue* de um elemento nas restantes situações.

$$deq' = mult \cdot tot (deq \cdot !flush') (\neg \cdot (\widehat{\wedge}) \cdot (empty \times empty) \cdot \pi_1)$$

Por fim é representada a nossa *queue*, expondo os métodos que esta disponibiliza através da função *sum3* (de salientar que o *flush* não é exposto para o utilizador, tratando-se de uma operação intermédia interna da máquina) :

$$queue = sum3 \text{ enq}' \text{ deq}' \text{ peek}'$$

3 Faulty Queue

Considerando como ponto de partida a *stack probabilística* das aulas, foi construída a *faulty queue* através da introdução de probabilidades de sucesso nos métodos que constituem a *queue* apresentada acima:

```
skip' = return · (id × bang)
peek'' p = MbT · schoice p peek' peek'
enq'' p = MbT · schoice p enq' skip'
deq'' p = MbT · schoice p deq' peek'
queue_p p q = sum2 (enq'' p) (deq'' q)
```

4 Testes e Resultados

Segue-se um teste que testa o funcionamento da *Faulty Queue*, utilizando outra função desenvolvida nas aulas, *toSMT*.

```
toSMT m = SMT · (curry (fmap swap · m · swap))
t'' = do
  x ← toSMT (peek'' 0.8) ()
  y ← toSMT (deq'' 0.9) ()
  toSMT (enq'' 0.6) (x + y)
  toSMT (deq'' 0.7) ()
```

O resultado obtido, para a execução apresentada, foi o seguinte:

```
$ runSMT t'' ([1, 2, 3], [4, 5, 6])
Just 2 90.0 %
Just 1 10.0 %
```

Podemos concluir que, na nossa implementação pelo menos, não será possível obter dados sobre a realização de *flush* pois trata-se de um método interno da máquina.