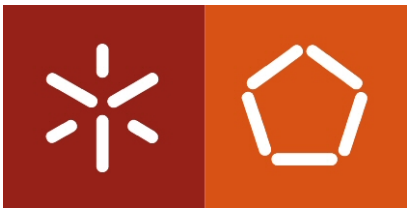

Relatório do Projecto de Laboratórios de Informática I (2012/2013)

64315 - Filipe Ribeiro
64288 - Nelson Gomes

Jan. 2012

Resumo

Este relatório descreve as varias etapas pedidas no âmbito deste mesmo projeto. Serão demonstradas e explicadas oito das nove pedidas. Cada etapa esta dividida em varias partes, o que e pertendido que a função faça, o tipo, funções auxiliares, código Haskell e a análise ao cógido.

1 Programação

1. Implementar a função $idsOk :: House- > Bool$ que verificar que não há identificadores repetidos (nas duas listas).

   - Tipo de dados:

     ```
     idsOk :: House -> Bool
     ```

     O objetivo e ao receber um $House$ teste ($Bool$) se de todos os identificadores existem repetidos.

   - Codigo Haskell:

     ```
     idsOk :: House -> Bool
     idsOk h = let l = hrunning h ++ hfinished h
                   m = nub(map actid l)
               in  length m == length l
     ```

   - Análise do código:

     O let cria duas variáveis, l e m, a primeira cria uma lista com os artigos que estão no $hrunning\ h$, e no $hfinished\ h$. A segunda, utiliza duas funções predefinidas do Haskell nub[1] e map[2].

     ```
     ... (map actid l)
     ```

     O map cria uma lista de todos os $actid$ pertencentes há lista l, que se construiu antes.

     ```
     ... m = nub(map actid l)
     ```

     O nub tem um papel importante, ao receber a lista vinda do map, constroi outa lista com os números pertencentes há lista que recebe, e se existir repetidos exclui esses mesmos. Logo a comparação final,

     ```
     ... in  length m == length l
     ```

     como mostra no código, se o comprimento de l (lista original), for igual ao comprimento da lista m (não existir identificadores repetidos) a função toma o valor lógico $True$ e se não for o caso, $False$.

[1] nub :: $Eq\ a => [a]- > [a]$
[2] map :: $(a- > b)- > [a]- > [b]$

2. Acrescentar a funcão $idKey :: House- > Bool$ que verifica se o identificador $actid$ de um dado item o identifica univocamente.

   - Tipo de dados:

     ```
     idKey :: House -> Bool
     ```

     O objetivo é, ao receber um $House$ teste ($Bool$) se o identificador de um dado item o identifica univocamente.

   - Código Haskell:

     ```
     idKey :: House -> Bool
     idKey h = let l = hrunning h ++ hfinished h
                   m = idsOk h
               in if (m == False) then verifica l else True

     verifica :: [Auction] -> Bool
     verifica [] = True
     verifica (h:t) | ajuda h (h:t) = verifica t
                    | otherwise = False

     ajuda :: Auction -> [Auction] -> Bool
     ajuda _ [x] = True
     ajuda a t = let b = nub(filter((==actid a).actid) t)
                 in length b == 1
     ```

   - Análise do código:

     Nesta função também o let cria duas variaveis l e m, o l é novamente uma lista com os artigos que estão no $hrunning\ h$, e os que estão no $hfinished\ h$, mas m chama a função anterior, $idsOk$, visto que se $idsOk$ toma o valor lógico de $True$, que diz que não ha identificadores repetidos, a função que estamos a analisar toma também o valor de $True$ e termina.

     Caso a função $idsOk$ toma o valor lógico $False$ a função em estudo chama uma outra função, uma função auxiliar:

     ```
     verifica :: [Auction] -> Bool
     verifica [] = True
     verifica (h:t) | ajuda h (h:t) = verifica t
                    | otherwise = False
     ```

     Esta função ao receber uma lista de $Auction$ vai comparar recursivamente o identificador de cada item com o do resto dos itens e verifica se existe outro igual, para isso recorre a outra função:

     ```
     ajuda :: Auction -> [Auction] -> Bool
     ajuda _ [x] = True
     ajuda a t = let b = nub(filter((==actid a).actid) t)
                 in length b == 1
     ```

Esta função auxiliar tém um papel simples mas importante, dado um $Auction$ e $[Auction]$ testa se existe algum identificador igual ao do $Auction$ na $[Auction]$, na sua execução cria uma variavel, b, que utiliza duas funções já existentes no Haskell, nub e $filter$[3] o resultado da aplicação desta ultima função,

```
... (filter((==actid a).actid) t)
```

cria uma lista, que indo aos $actid$ de t, seleciona os que são iguais ao $actid$ de a, depois com o nub, se o comprimento da lista dada for um, significa que so existe um, o proprio $actid\ a$, podendo-se deduzir que de facto não existe identificadores repetidos, tomando a função $ajuda$ o valor de $True$.

Voltando há função $verifica$, se o resultado da função $ajuda$ for $True$ volta a fazer o mesmo procedimento mas desta vez aplicado só há cauda da lista.

```
... | ajuda h (h:t) = verifica t
```

Caso contrário o resultado da função $verifica$ é $False$, o que quer dizer que existe um identificador de um dado item que não o identifica univocamente, tomando também a função principal o valor de $False$.

3. Será que $idKey$ implica $idsOk$? Ou será que $idsOk$ é que implica $idKey$? Ou nenhuma dessas implicações se verifica? Justifique a sua resposta.

Na nossa opinião $idKey$ implica $idsOk$ pelo facto de, sabendo que $idsOk$ toma o valor lógico True então $idKey$ toma de imediato o mesmo valor de $True$ e sendo o contrario, havendo $actid$ repetidos é que $idKey$ faz a sua verificação.

4. Acrescentar o predicado $allIds :: House -> Bool$ que verifica se a alocação de identificadores a leilões é sequencial.

   - Tipo de dados:

     ```
     allIds :: House -> Bool
     ```

     O objetivo e ao receber um $House$, teste ($Bool$) se os identificadorres de um leilão estao de forma sequencial, quer seja crescente quer decrescente.

[3] filter :: $(a -> Bool) -> [a] -> [a]$

- Código Haskell:

```
allIds :: House -> Bool
allIds h =
  let (a,b)=((map actid (hrunning h)),(map actid (hfinished h)))
  in (abOk a b)

abOk :: [Int] -> [Int] -> Bool
abOk x y = let m = f x
               n = g x
               m1 = f y
               n1 = g y
           in (m || n) && (m1 || n1)

f :: [Int] -> Bool
f [] = True
f [x] = True
f (h:x:xs) = (h>=x)&&(f (x:xs))
g :: [Int] -> Bool
g [] = True
g [x] = True
g (h:x:xs) = (h<=x)&&(g (x:xs))
```

- Análise do código:

  A função principal,

```
allIds :: House -> Bool
allIds h =
  let (a,b)=((map actid (hrunning h)),(map actid (hfinished h)))
  in (abOk a b)
```

  constroi duas variáveis, crindo o par, (a, b), a primeira, usando o map cria uma lista com todos os $actid$ pertencentes ao $hrunning\ h$, enquanto a segunda, cria também uma lista de $actid$, mas desta vez pertencentes ao $hfinished\ h$.

  Depois de criada as variáveis, a função chama uma função auxiliar $abOk$,

```
abOk :: [Int] -> [Int] -> Bool
abOk x y = let m = f x
               n = g x
               m1 = f y
               n1 = g y
           in (m || n) && (m1 || n1)
```

  esta função, verifica se tanto os $actid$ de a e b estão de alguma forma ordenados. Em comparação com a função principal, esta também utiliza outras funções auxiliares:

$f :: [Int] -> Bool$ | $g :: [Int] -> Bool$
$f[] = True$ | $g[] = True$
$f[x] = True$ | $g[x] = True$
$f(h : x : xs) = (h >= x)\&\&(f(x : xs))$ | $g(h : x : xs) = (h <= x)\&\&(g(x : xs))$

Estas duas funções tem um papel minucioso, a função f verifica se a lista que recebe é decrescente e a g se é crescente. Guardando os valores dos testes de cada uma das listas em quatro variaveis, m, n, $m1$, $n1$.

Voltanto há função principal, a operação final,

```
... (m || n) && (m1 || n1)
```

sendo que as listas só precisão de estar ordenadas de uma das formas, dai a condição $(||)$, mas independentemente da ordenação, precisão de estar ordenadas, dai a condição $(\&\&)$.

5. É fácil de ver, exercitando O$_{\text{L}}$X$^{\text{UM}}$, que a função $auctionBid$ não está implementada convenientemente, pois:

   - após fazer Reset, se licitar o item 1 por 61 euros e depois por 1 (um) euro, verá que o sistema aceita qualquer oferta, mesmo que seja inferior à melhor oferta até ao momento, violando o princípio do *quem dá mais ganha* inerente a qualquer leilão;
   - se licitar um item que não existe, por exemplo o item 99, o sistema dá erro.

   Reescreva a função por forma a nenhuma das anomalias identificadas acima se verificar.

   - Versão melhorada: Tipo de dados:

     ```
     auctionBid :: House -> Int -> String -> Int -> House
     ```

     O objectivo e aumentar o rigor do leilão, ao receber um $House$, o identificador do item, o nome de quem esta a tentar comprar e o valor que oferece pelo produto.

   - Versão melhorada: Codigo Haskell:

```
auctionBid :: House -> Int -> String -> Int -> House
auctionBid h id bidder bid =
 let curr = (filter ((==id).actid) (hrunning h))
 in if (curr == []) then error "Item não existente" else
        if (podee curr bid) then (add h curr id bidder bid)
              else error "Existe um valor de licitação superior"

podee :: [Auction] -> Int -> Bool
podee h i = let l = head(h)
            in i > (actvalue l)

add :: House -> [Auction] -> Int -> String -> Int -> House
add h curr id bidder bid =
  let r = filter ((/=id).actid) (hrunning h)
      cur = head(curr)
      newa = Auction (actid cur) (actowner cur) (actdesc cur)
                     (actclass cur) (actvalinc cur) bid bidder
  in House (newa:r) (hfinished h)
```

- Análise do código:

A grade diferença entre a versão antiga e a melhorada, é como foi referido, a rigor do leilão, dando avisos a quem esta a participar no leilão. Existem dois avisos possiveis,

```
... error "Item não existente" else
      ...
        ... error "Existe um valor de licitação superior"
```

o primeiro, e quando o utilizador pertende adquirir um item, colocando um identificador inixestente, o segundo e quando o valor oferecido pelo item e inferior a um já oferecido.

A primeira tarefa que a função $auctionBid$ faz e criar uma variavel, $curr$, usando o $filter$, cria uma lista de $Auction$, pertencentes ao $hrunning\ h$, com os que tenham $actid$ iduais ao id dado, e faz logo de seguida o teste, se a lista for vazia, que significa que não ha identificadores iguais, aparece o primeiro erro. Se não for esse o caso faz a verificação sobre o valor da licitação, chamando a função auxiliar:

```
podee :: [Auction] -> Int -> Bool
podee h i = let l = head(h)
            in i > (actvalue l)
```

Esta função testa se o valor oferecido e superior ao valor já anteriormente oferecido pelo item, a utilização do $head$, apesar de a lista h ter normalmete um elemento, é para ter selecionado um elemento e não uma lista, para facilitar o teste seguinte.

```
... in i > (actvalue l)
```

Esta compara o valor a que o utilizador está a oferecer pelo item com o valor já existente de licitações anteriores, se for não for maior aparece o segundo erro.

Se tiver passado por estes dois testes, é precisso registar o licitação, chamando assim outra função,

```
add :: House -> [Auction] -> Int -> String -> Int -> House
add h curr id bidder bid =
 let r = filter ((/=id).actid) (hrunning h)
     cur = head(curr)
     newa = Auction (actid cur) (actowner cur) (actdesc cur)
                    (actclass cur) (actvalinc cur) bid bidder
 in House (newa:r) (hfinished h)
```

Esta função ao receber todos os dados de uma licitação, regista-os no $House$ que recebe. Cria a variavel r, que indo aos $actid$ dos $Auction$ que recebe da lista $hrunning\ h$ cria uma lista dos $Auction$ cujo $actid$ sejam diferentes do id dado na função. Cria também o cur que é o item que o utilizador esta a licitar, e cria um novo $Auction$ com os dados iguais ao do cur juntando o valor licitado e o nome do comprador. Juntando no final toda a imformação no $House$, e junta a lista $hfinished\ h$.

6. Supondo que o formato interno $House$ muda para

```
data NHouse = NHouse { tot :: [ NAuction ] } deriving (Show,Eq)
```

onde

```
data NAuction = NAuction { a :: Auction, st:: Status }
                                                deriving (Show,Eq)

data Status = Running | Finished deriving (Eq,Show)
```

escrever funções de conversão entre os dois formatos,

- $toNHouse :: House -> NHouse$
- Código Haskell:

```
toNHouse :: House -> NHouse
toNHouse h = let a = (addd (hrunning h) (Running))
                 b = (addd2 (hfinished h) (Finished))
             in NHouse (a ++ b)

addd :: [Auction] -> Status -> [NAuction]
addd [] _ = []
addd (h:t) a = NAuction (h) (Running) : addd t a

addd2 :: [Auction] -> Status -> [NAuction]
addd2 [] _ = []
addd2 (h:t) a = NAuction (h) (Finished) : addd2 t a
```

- Análise do código:

O $NHouse$ é uma lista composta por $Auction$, e por $Status$, este que define o estado do item, $Running$ ou $Finished$.

A função $toNHouse$ primeiro cria duas variáveis, que são as listas a e b, a primeira usando a função auxiliar $addd$.

```
addd :: [Auction] -> Status -> [NAuction]
addd [] _ = []
addd (h:t) a = NAuction (h) (Running) : addd t a
```

Esta função ao receber uma lista de $Auction$ e o estado, cria uma lista de $NAuction$ com cada item da lista h e com o estado, neste caso $Running$.

A variavel b usa a função auxiliar $addd2$ que que é muito parecida com a $addd$, com a diferença que recebe o estado como $Finished$.

```
addd2 :: [Auction] -> Status -> [NAuction]
addd2 [] _ = []
addd2 (h:t) a = NAuction (h) (Finished) : addd2 t a
```

Por fim, a função principal ao receber as duas listas, junta-as criando um $NHouse$.

```
... in NHouse (a ++ b)
```

- $toHouse :: NHouse -> House$

- Código Haskell:

```
toHouse :: NHouse -> House
toHouse nh = let r = (map a ((filter ((==Running).st) (tot nh))))
                 f = (map a ((filter ((==Finished).st) (tot nh))))
             in House (r) (f)
```

- Análise do código:

Esta função tem como objetivo fazer o contrário da anterior. Esta, fazendo uma "triajem"aos estados de cada item facilmente os consegue separar para criar o $House$ pertendido. Para isso função cria duas variáveis r e f a primeira, vai a lista recebida, nh, que e um tot, e cria outra lista cujos os estados dos itens sejam $Running$, essa lista é depois usada no map que recebe também o a que, segundo a definição do $NAuction$, é o $Auction$.

```
... r = (map a ((filter ((==Running).st) (tot nh))))
```

A segunda variavel usa novamente o map e o $filter$, mas desta vez, aplicado aos itens com o estado $Finished$, criando também uma lista desses mesmos itens.

```
... f = (map a ((filter ((==Finished).st) (tot nh))))
```

Por ultimo, a função principal, a partir das listas r e f constroi o $House$:

```
... in House (r) (f)
```

7. Redefinir o tipo $Auction$ acrescentando-lhe mais um campo — $actclass$,

```
data Auction =  Auction {
                  actid     :: Int,
                  actowner  :: String,
                  actdesc   :: String,
                  actclass  :: String,
                  actvalue  :: Int,
                  actbidder :: String
                } deriving (Show,Eq,Ord)
```

que classifica itens em categorias (eg. electro-doméstico, mobiliário) e escrever uma nova função de administração que totaliza os valores leiloados por categoria (x de electro-domésticos, y de mobiliário, etc).

- Tipo de dados:

```
tClass :: House -> [TotClass]
```

Recebendo um $House$ devolve uma lista de $TotClass$. Mas o que é um $TotClass$?

```
data TotClass = TotClass {classe :: String, valor :: Int}
                                                deriving Show
```

Definimos um $TotClass$ sendo composto por $classe$ e por $valor$, este que totaliza os valores dos itens leiloados de uma determinada $classe$.

- Código Haskell:

```
tClass :: House -> [TotClass]
tClass h = let l = hrunning h ++ hfinished h
               q = nub(map actclass (l))
           in lista l q

lista :: [Auction] -> [String]-> [TotClass]
lista l [] = []
lista l (h:t) = let a = filter((==h).actclass)l
                    b = map (actvalue) (a)
in TotClass (h) (foldr (+) 0 b):(lista l t)

data TotClass = TotClass {classe :: String, valor :: Int}
                                                deriving Show
```

- Análise do código:

A função principal começa como muitas outras por criar duas variaveis, a primeira l, como já tinhamos visto anteriormente cria uma lista com os artigos que estão no $hrunning\ h$, e os que estão no $hfinished\ h$. A segunda, q, usando o map para que, indo aos $actclass$ dos itens lista l crie uma lista com todas as classes existentes na lista l. Em seguida a função nub ao receber essa lista e elimina, caso exista, todas as classes repetidas.

```
... q = nub(map actclass (l))
```

De seguida chama a função auxiliar $lista$. Esta função é que faz o trabalho principal, tendo também duas variaveis, a primeira, desta vez a, usando o $filter$ cria um lista que recorrendo á lista recebida, l, vai pegar os item com a classe igual as classes recebidas na lista.

```
... a = filter((==h).actclass)l
```

A segunda variavel, b, usando o map, pega na lista a e de cada item vai buscar o seu valor, $actvalue$.

```
... b = map (actvalue) (a)
```

Por ultimo a função $lista$, para construir o $TotClass$, interpreta o h como sendo a classe e usando uma função existente em Haskell, $foldr$[4], que recebendo a instrução de soma, o caso de paragem, neste caso o elemento neutro da soma, e a lista vinda de b, soma todos os valores dos artigos com classes que estejam na lista recebida, calculando com o uso a recursivamente. Formando assim um $TotClass$ para cada classe.

```
... in TotClass (h) (foldr (+) 0 b):(lista l t)
```

8. Um dos defeitos do modelo de dados $Auction$ é não guardar o valor inicial de licitação. Corrijir esta limitação, criando ainda uma função que contabilize quanto é que, em média, os itens vendidos se valorizaram no leilão.

   - Tipo de dados:

```
media :: House -> (Float, String)
```

   Para esta função acrescentamos ao tipo $Auction$ outro campo "actvalinc", que guarda o valor inicial. O resultado desta função é, como diz no timo, $(Float, String)$, sendo o $(Float)$ a diferença média dos artigos, e consuante esse valor, exprime na $String$ de valurizou ou não valurizou.

   - Código Halkell:

```
media :: House -> (Float, String)
media h = let l = hfinished h
              a = map actvalinc l
              b = map actvalue l
          in auxMedia (zip a b)

auxMedia :: (Fractional t, Integral a, Ord t)=>[(a, a)]->(t, String)
auxMedia x = let a = map (\(m,n) -> n-m)x
                 b = fromIntegral (sum a)
                 c = fromIntegral (length a)
                 d = b/c
             in if (d>0) then (d,"Valorizou")
                         else (d,"Nao valorizou")
```

[4] foldr :: $(a->b->b)->b->[a]->b$

- Análise do Código:

A função principal, cria três listas como variáveis, l que como os itens vendidos, a lista l só utiliza o $hfinished\ h$. A segunda, a, e a treceira ,b, são semelhantes, mas a primeira cria uma lista dos valores iniciais, $actvalinc$, enquanto a segunda cria dos valores a que os produtos foram vendidos, $actvalue$. Chando de seguida a função auxiliar $auxMedia$, que atravez da função zip[5] recebe uma lista de pares, com o valor inicial, e o valor final.

A função auxiliar,

```
auxMedia :: (Fractional t, Integral a, Ord t)=>[(a, a)]->(t, String)
auxMedia x = let a = map (\(m,n) -> n-m)x
                 b = fromIntegral (sum a)
                 c = fromIntegral (length a)
                 d = b/c
             in if (d>0) then (d,"Valorizou")
                         else (d,"Nao valorizou")
```

tem como objetivo tratar das operações. Esta atravéz do let faz grande parte do trabalho, a variável a resulta de aplicar o map com uma função anónima, que pegando na lista que a função recebe como argumento, faz a subtração do valor final pelo valor inicial. Nas duas seguintes, b e c, a primeira faz o sumatório dos valores resultantes de a, e a segunda calcula o comprimento da lista a. A utilização do $fromIntegral$ é para mais tarde poder utilizar a barra de divisão. A última, d, apenas faz a operação de subtração.

Por último a função faz uma pequena verificação, se o valor da média for maior que zero, significa que o produto valorizou, caso contrário não valorizou.

```
... in if (d>0) then (d,"Valorizou") else (d,"Nao valorizou")
```

[5] $zip :: [a] -> [b] -> [(a, b)]$