Métodos Formais de Programação I +
Opção - Métodos Formais de Programação I

4.º Ano da LMCC (7007N2) + LESI (5307P6)
Ano Lectivo de 2001/02

Exame (época especial) — 13 de Novembro 2002
14h00
Sala 2203

NB: *Esta prova consta de 7 alíneas todas com a mesma cotação*

PROVA SEM CONSULTA (2 horas)

Questão 1 Considere o seguinte fragmento de VDM-SL:

```
ordered: seq1 of int -> bool
ordered(l) ==
    cases l:
      [e]     -> true,
      others  -> hd l < Max(tl l) and ordered(tl l)
    end;

max: int * int -> int
max(i,j) == if i >= j then i else j;

Max: seq1 of int -> int
Max(l) ==
    cases l:
      [e]     -> e,
      others  -> max(hd l, Max(tl l))
    end;
```

1. Mostre, identificando o respectivo tipo indutivo e desenhando os respectivos diagramas, que Max é um catamorfismo e que $ordered$ é um paramorfismo.
2. Aplique a $ordered$ a lei de Fokkinga para obter como resultado uma nova versão dessa função baseada num catamorfismo e não num paramorfismo.
3. Pretendendo especificar uma função $sort$ tal que $ordered(sort\, l) = true, \forall l$, alguém escreveu

   ```
   sort: seq1 of int -> seq1 of int
   sort(l) == [ l(n) | n in set inds(l) & l(n) <= l(n+1) ];
   ```

   Infelizmente, esta função tem um problema de indefinição. Qual? Mostre ainda que, mesmo corrigindo esse problema, a função não satisfaz a equação dada.

Questão 2 Mostre que a operação de inversão de listas

```
invl[@A] : seq of @A -> seq of @A
invl(l) == if l = [] then l else invl[@A](tl l) ^ [hd l] ;
```

preserva índices, isto é, que

$$inds \cdot invl = inds \tag{1}$$

se verifica, onde

$$inds = inseg \cdot len$$

Sugestão: pode fazê-lo por indução em listas ou por fusão-cata.

Questão 3 Responda às seguintes 3 alíneas de uma das fichas das aulas práticas desta disciplina:

Hash tables are well known data structures whose purpose is to efficiently combine the advantages of both static and dynamic storage of data. Static structures such as *arrays* provide random access to data but have the disadvantage of filling too much primary storage. Dynamic, *pointer*-based structures (*e.g.* search lists, search trees *etc.*) are more versatile with respect to storage requirements but access to data is not as immediate.

The idea of *hashing* is suggested by the informal meaning of the term itself: a large database file is "hashed" into as many "pieces" as possible, each of which is randomly accessed. Since each sub-database is smaller than the original, the time spent on accessing data is shortened by some order of magnitude. Random access is normally achieved by a so-called *hash function*,

$$Data \xrightarrow{hash} Location$$

which computes, for each data item, its location in the *hash table*. Standard terminology regards as *synonyms* all data competing for the same location. A set of synonyms is called a *bucket*. There are several ways in which data collision is handled, *e.g. linear probing* or *overflow handling*. Below we consider the latter.

1. Write and validate a VDM-SL model $HTable$ for hash-tables (with overflow handling) in which you should model (a) locations as integers; (b) collision buckets as finite sets of (data) synonyms; (c) hash tables as mappings from locations to collision buckets.
2. Assuming some predefined hash function $hash$, add to $HTable$ an invariant stating that all data in the same collision bucket share the same hash-index. Can you infer from this invariant that buckets are mutually disjoint?
3. Specify the following functionality on top of $HTable$ so that your invariant (above) is preserved:

```
Init : () -> HTable

Insert : Data * HTable -> HTable

Find   : Data * HTable -> bool

Remove : Data * HTable -> HTable
```