Advanced Functional Programming WS 2025/2026 LVA 703139

Exercise Sheet 6, 10 points Deadline: Tuesday, November 18, 2025, 4pm

- Solve the tasks in files `Exercise06_*.hs` and upload only these files in OLAT.
- Mark the solved exercises in OLAT.
- Your modified `Exercise06_*.hs` files must compile with `ghci` without error messages.

Task 1 *Embedding Relation* **6 p.**

1. Improve the monadic implementation, so that there is no significant time difference between all four presented variants, i.e., using `ST` or `State` with or without computation of size.

   To this end, think about how to make Boolean operations (disjunction, conjunction, all, any) lazy in their monadic versions. In particular, the size of the final map should become significantly smaller by your optimizations. (3 points)

2. Use the labeling of terms that has been developed in the exercise of the previous week. Change the type of the dictionary so that labels are used as keys instead of the full terms.

   What changes? (3 points)

Task 2 *Tseitin* **4 p.**

The function `tseitin` runs in quadratic time. The reason is that the writer part of the currently used monad `RWS ... [Clause] ...` uses lists in the output part, and each `tell cl` will append a clause `cl` to the end of the output list. So standard lists are not a good choice as the `w`-parameter for a writer in this application.

Note that `w` can be an arbitrary `Monoid`, cf.

https://hackage.haskell.org/package/mtl/docs/Control-Monad-Writer-Lazy.html.

In this exercise, an alternative monoid should be used, namely one where we store the monoid operations in a tree.

```
data OpTree a =
    Append (OpTree a) (OpTree a)
  | Singleton a
  | Empty
```

1. Make `OpTree a` an instance of `Monoid`. (1 point)
2. Implement `monoidToList :: OpTree a -> [a]` in an obvious way, which most likely will result in a quadratic algorithm. (1 point)
3. Redesign `monoidToList` so that it runs in linear time. (2 points)

After your modifications of the program, `testInvocation` should produce the same CNF as before, i.e., the upcoming text.

```
ghci> testInvocation
c  variable "1" is encoded as number 4
c  variable "a" is encoded as number 3
c  variable "c" is encoded as number 1
p cnf 7 12
7 0
2 1 0
-2 -1 0
5 4 0
-5 -4 0
-6 3 5 0
6 -3 0
6 -5 0
7 -2 -3 -6 0
-7 2 0
-7 3 0
-7 6 0
```

Moreover, running `testPerformance n` should be linear in `n`, if all three tasks have been implemented correctly. For instance, on a test machine, the list-based version requires 1 second, 4 seconds, and 9 seconds to evaluate `testPerformance 10000`, `testPerformance 20000`, and `testPerformance 30000` respectively, illustrating quadratic runtime.
If one switches to the `OpTree` monoid with the linear version of `monoidToList`, then the new timings might be 0.09 seconds, 0.19 seconds and 0.26 seconds, respectively, i.e., a linear behavior is visible, and the overall runtime is clearly improved.