universität innsbruck

Advanced Functional Programming | WS 2025/2026 | LVA 703139

Exercise Sheet 11, 10 points | Deadline: Tuesday, January 13, 2026, 4pm

- Solve the tasks in file `Exercise11.hs` and `Exercise11_SMT.hs` upload only these files in OLAT.
- Mark the solved exercises in OLAT.
- Your modified `.hs`-files must compile without error messages.

Task 1 *Bracket-Version for Asynchronous Actions* **2 p.**

If one spawns threads for asynchronous actions then there are two aims:

1. no exception in spawned threads is silently swallowed
2. no thread is left running unintentionally

In the lecture you have seen the code

```
do a1 <- async act1
   a2 <- async act2
   r1 <- wait a1
   r2 <- wait a2
   return (r1, r2)
```

Task: Figure out which of the properties (1) or (2) may be violated in the presence of exceptions. Give a concrete scheduling how things can go wrong.

Remark: Because of this problem, there is a bracket-version of `async` that takes care of all clean-up (similarly to `withFile`, `withTempFile`, ...):

```
withAsync :: IO a -> (Async a -> IO b) -> IO b
```

Most often, you want to use this function instead of the low-level function `async` itself.

Task 2 *firstJust* **4 p.**

We are interested in parallel execution of various actions, where we search for the first successful result. To this end, your task is to implement the following easy-to-use high-level function:

```
firstJust :: [IO (Maybe a)] -> IO (Maybe a)
```

It should have the following behavior:

- all IO actions are executed in parallel
- the first (w.r.t. execution time) IO action that returns a `Just`-result is taken as final result
- if no IO action returns a `Just`-result, then `Nothing` is returned
- exceptions of the actions should be ignored
  (these actions are logically equivalent to result in `Nothing`)
- as soon as the first `Just`-result is received, the execution of all other actions should be stopped
- if the `firstJust`-thread itself is interrupted by some asynchronous exception, then all spawned threads should be cancelled as well.

Hints: preferably use `withAsync` for the creation of `ASync`s, use some variant of `wait` for waiting on a result, and use some variant of `cancel` for stopping the executions. See https://hackage.haskell.org/package/async/docs/Control-Concurrent-Async.html#g:8 for possibilities. In particular, do not use something like `bracket <spawn> <cancel> <doSomething>`, as this pattern does not work for spawning threads.

You can execute `cabal run Exercise11 -- haskell +RTS -N4 -RTS` to start some application that runs 5 worker threads in parallel. As soon as one of them reports "success", the program should immediately report "result of firstJust: ..." and then you have to press enter twice to quit, cf. `mainTask2and3`. If you hit enter before the first successful computation, then all threads should be aborted immediately and the result of `firstJust` should be `Nothing`.

Task 3 *Killing External Computations* **2 p.**

In order to cancel computations, it might also be the case that you need to cancel externally spawned processes, e.g., some SMT-solver.
Change the `workerExternal`-code, so that it is killing its spawned process, if the thread is interrupted by some exception.
In order to test the application,

- once run `ghc --make -O2 fibExtern.hs` to obtain some executable
- on demand, adjust the name of the executable in the file (e.g., by adding .exe)
- for testing, invoke `cabal run Exercise11 -- extern +RTS -N4 -RTS`

After completion of this task, after running the test application, there should be no fibExtern programs still running. (if you did not kill all processes, you might want to manually kill running fibExtern instances via some OS command, e.g. `killall fibExtern`)

Task 4 *Putting everything together: Strategies for Termination Proving* **2 p.**

Design terminationProver in a way that it searches in parallel for either

- some LPO (`lpoSolver`) or
- some LPO with lexicographic comparison from right-to-left (`reverseLpoSolver`).

Both of these functions are available as black-boxes which are of no further interest in this exercise.
The search should be aborted after the given timeout, or if some LPO has been found.
You can test your implementation with a 5 seconds timeout, e.g., by running

```
cabal run Exercise11 lpo 5 ariTRSs.txt -- +RTS -N4 -RTS
```

Depending on the speed of your machine, you should find 74 proofs.
Independently of the speed, it should always be the case that

- the `Exercise11` process causes at most 200 % CPU load
  (so all worker threads are stopped after each example TRS)
- there are at most two `z3` processes running
  (so all spawned processes are stopped after each example TRS)

In order to ensure the latter, you also need to refactor `runSmtSolver` in `Exercise11_SMT.hs` in the style of Task 3.