universität innsbruck

WS 2024/2025

Advanced Functional Programming

Week 12 – Profiling, Efficient Data Structures

René Thiemann

Department of Computer Science

Last Week

- channels
  - more high level interface for a concurrent data-structure; implementing double ended linked lists using `MVar`s
- asynchronous actions
  - perform IO-actions asynchronously
  - wait on tasks to complete
  - various versions of `async`-library
- asynchronous exceptions
  - cancellation of tasks
  - bracket-construct (or with...-construct) to safely close files, kill external processes, etc., even in case of asynchronous exceptions

Profiling

Profiling

- profiling is a method for analyzing runtime behavior
- aim: get detailed statistics about time and space usage to facilitate performance tuning
- workflow of profiling
  1. identify cost centers, i.e., functions for which data should be reported
  2. run program in profiling mode
  3. inspect generated profiling statistics and identify hot-spots
  4. study hot-spots and try to optimize these parts
  5. go back to step 2

Step 1: Annotate Cost Centers in Haskell

- two ways of annotations
- manual annotation
  - use annotation `{-# SCC "name" #-}` in front of some expression (SCC = Set Cost Center)
  - resource consumption of running this expression will then be added to the profiling statistics, tagged with `"name"`
- automatic annotation
  - adds cost centers for a selection of functions
  - automatic annotation is triggered via ghc-flags or cabal-flags

Step 2: Run Program in Profiling Mode in Haskell

- in Haskell, profiling has to be activated both at compile-time and at run-time
- compile-time
  - `ghc`: use ghc-options `-prof` (and on demand `-fprof-late` or other options for automatic annotations)
  - `cabal`: use cabal-options `--enable-profiling` and further options for automatic annotations
- run-time
  - add runtime system parameter `-p`
  - obtain `executable.prof` file that contains profiling statistics

Example LPO: Steps 1 and 2

- take solution of Exercises of week 11 (concurrent termination prover via LPO)
- run via cabal with profiling enabled

```
cabal run Exercise11 --enable-profiling --profiling-detail late --
  lpo 5 ariTRSs.txt
  +RTS -p
```

  - detail-level (automatic cost centers): `default`, `none`, `exported-functions`, `all-functions`, `toplevel-functions`, `late`
- observations
  - activation of profiling is easy, in particular with automatic cost center annotations
  - warning: profiling code may change optimization phase of ghc, so the behavior of profiled code might be different from original code
    - use `all-functions` and `toplevel-functions` with care
    - `late` inserts profiling code after optimization and therefore is recommended automatic mode (requires ghc $\geq$ 9.4.1)
    - disadvantage of `late`: names of functions after optimization are used

Example LPO: Step 3 – Investigate Profiling Statistics

- after execution inspect file `Exercise11.prof`, (some lines and columns deleted)

```
COST CENTRE          SRC                           %time %alloc

$fOrdTerm_$ccompare  Demo09_Parser_ARI.hs:6:56-58   72.6    0.0
$fOrdTerm            Demo09_Parser_ARI.hs:6:56-58   18.7    0.0
$wlpoEncoder         <no location info>              1.7   13.5

COST CENTRE            SRC                             entries  %time %alloc   %time %alloc

MAIN                   <built-in>                            0    0.3    0.1   100.0  100.0
 main                  Exercise11.hs:230:1-4                 1    0.0    0.0    99.7   99.9
   lpoSolver1          <no location info>                  566    0.0    0.0    50.1   48.5
    $wrunSmtSolver     <no location info>                  566    0.6    0.3    50.1   48.5
         $wlpoEncoder  <no location info>               646839    0.9    6.7    47.7   10.2
          $wgo15       <no location info>             10118272    0.3    0.0    30.4    0.0
           ccompare    Demo09_Parser_ARI.hs:6:56-58  452806139   23.8    0.0    30.1    0.0
          $w$sgo15     <no location info>              5580206    0.3    2.0    16.2    2.0
           ccompare    Demo09_Parser_ARI.hs:6:56-58  238906955   12.6    0.0    15.9    0.0
   reverseLpoSolver1   <no location info>                  566    0.0    0.0    49.3   48.4
```

Example LPO: Steps 3 and 4 – Identify Hot-Spots and Analyse

- explanations
  - the first list of cost centers are rankings of overall functions that cause the costs
  - the second list of cost centers is a tree like view
  - the obscure names are a result of `late`, use `toplevel-functions` or manual cost center annotations for improved readibility
  - the first time/alloc columns are costs that are caused by the current cost center
  - the second time/alloc columns are accumulated costs
- important: external costs do not occur in the data, e.g., cost of running SMT solver
- analysis of hot-spots
  - comparison of terms is the most costly operation
  - it is used for lookups in the dictionary to perform memoization in the LPO encoder
  - consequence: improve lookups (use integer-index or hash-maps or ..., cf. exercise)

Another Example: Computation of Mean

- computing the mean of a list of doubles, main function computes `mean [1..d]`

```
main = ... -- read d via getArgs and print "mean [1..d]"
mean1 :: [Double] -> Double
mean1 xs = ({-# SCC "sum" #-} sum xs)
  / fromIntegral ({-# SCC "len" #-} length xs)
```

- running the demo (1 = select mean1 function, 1e8 is value of d) with statistics

```
cabal run Demo12 -- 1 1e8 +RTS -s
```

- result: observe high memory consumption of 5.7 GB
- use profiling to trace memory usage over time via flag -hc

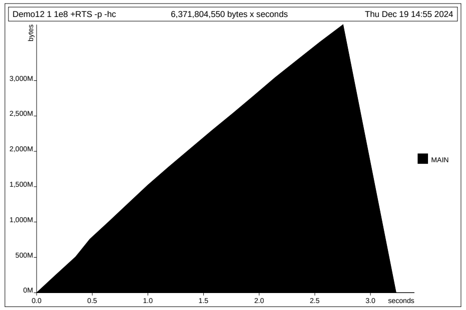
```
cabal run Demo12 --enable-profiling --profiling-detail none
  -- 1 1e8 +RTS -p -hc
```

- inspect generated data of file `Demo12.hp` via

```
hp2ps Demo12.hp && ps2pdf Demo12.ps && open Demo12.pdf
```


Demo12.hp – mean1

- obtain graph



- code analysis
  - generated list `[1..d]` is completely constructed in memory
  - problem: list elements are generated one-by-one for summation, and list needs to be kept for computing its length
  - interpretation of graph: once the length computation starts, memory can be freed again

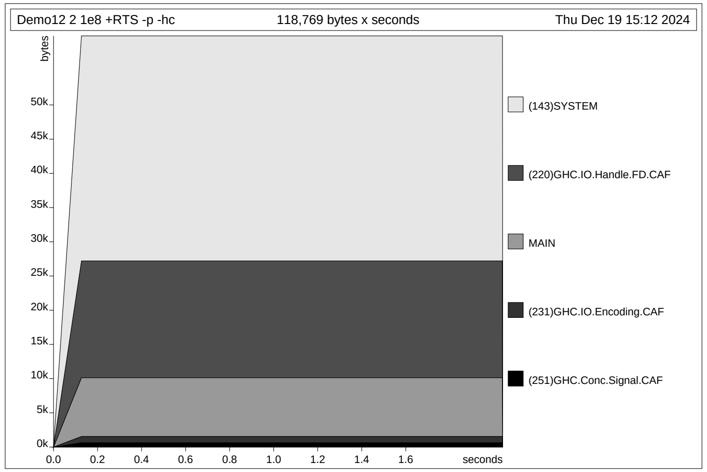
Solution: Compute Length and Sum in One Go

- optimized code for mean computation

```
mean2 :: [Double] -> Double
mean2 xs = let (s, l) = foldl' step (0, 0) xs in s / fromIntegral l
  where
    step :: (Double, Integer) -> Double -> (Double, Integer)
    step (s, l) x = let
        s' = s + x
        l' = l + 1
      in s' `seq` l' `seq` (s', l')
```

- use strict fold (`foldl'`) and `seq` to avoid generation of thunk in accumulator, e.g., 0 + x1 + x2 + x3 + ...
- both +`RTS -s` and +`RTS -hc -p` confirm improved memory usage (runtime is improved, too: less time required for garbage collection)

Demo12.hp – mean2



Efficient Data Structures

Choice of Data Structures

- efficiency is often obtained by choosing suitable data structures
- we consider two interesting scenarios
  - use mutable data structures within purely functional code
  - an example of a data structure designed for purely functional programming

Mutable State: Control.Monad.ST

- the monad `ST s` abstracts some type variable `s`, the state
- unlike the known `State s` monad, there is no way to access the full state (`getState`) or setting it (`putState`)
- instead, the state is just updated locally, e.g., by
  - creating a new reference or some mutable array
  - updating a reference, or some array content
- we have seen functionality like this already in `IO`, e.g., `newIORef`, `writeIORef`, etc.
- in contrast to `IO`, `ST s` can be used in purely functional code via `runST`
  - `runST :: (forall s. ST s a) -> a`
  - the universal quantifier ensures that no information of the state can leek into the `a`

Example: STRef in pure Code

- although `helloSTMain` uses mutable state, the result of `helloST` is pure

```
helloSTMain :: Int -> ST s (String, Int)
helloSTMain y = do
    s <- newSTRef "hello"
    x <- newSTRef y
    sVal <- readSTRef s
    modifySTRef x (+ 7)
    writeSTRef s (sVal ++ " world")
    sFin <- readSTRef s
    xFin <- readSTRef x
    return (sFin, xFin)

helloST :: Int -> (String, Int)
helloST y = runST (helloSTMain y)
```

- clearly, this could also have been done using the `State` monad, but see next slide

Example: Fibonacci Numbers via Mutable Arrays

- although `fibMain` uses mutable arrays, the results of `fib` and `fibArray` are pure

```
-- "STArray s indexType elementType" are mutable arrays living in state s
fibMain :: Int -> ST s (STArray s Int Integer)
fibMain n = do
  a <- newArray (0,n) 1  -- indices 0..n, array content initialized with 1
  mapM_ ( \ i -> do
    x <- readArray a (i - 2)
    y <- readArray a (i - 1)
    writeArray a i (x + y)) [2..n]
  return a

fibArray :: Int -> Array Int Integer -- Array: immutable pure arrays
fibArray n = runSTArray (fibMain n)  -- runSTArray freezes array

fib :: Int -> Integer
fib n = runST (do
  a <- fibMain n
  readArray a n)
```


Dedicated Functional Data Structures

- there are interesting data structures and algorithms targeting pure functional programming
  - non-destructible updates, i.e., immutable data
  - advantage: copying of these data structures is $O(1)$
- examples
  - finger trees (https://en.wikipedia.org/wiki/Finger_tree)
  - priority queues (https://en.wikipedia.org/wiki/Brodal_queue)
  - double ended queues (deques);
    queue version of Okasaki will be introduced on next slides

A First Simple Queue Implementation

- implementation uses two lists to represent queue:
  the beginning of the queue (`b`) and the end of the queue (`e`) in reverse order

```
data Queue1 a = Queue1 [a] [a]   -- Queue1 b e

empty1 :: Queue1 a
empty1 = Queue1 [] []

insert1 :: a -> Queue1 a -> Queue1 a
insert1 x (Queue1 b e) = Queue1 b (x : e)

remove1 :: Queue1 a -> (a, Queue1 a)
remove1 (Queue1 (x : b) e) = (x, Queue1 b e)
remove1 (Queue1 [] []) = error "empty queue"
remove1 (Queue1 [] e) = remove1 (Queue1 (reverse e) [])
```

- execution costs: worst case $O(n)$, amortized cost: $O(1)$

Tuning the Queue Implementation

- aim: get rid of expensive reverse operation
- main internal operation for queues: reverse e and append it to b
- idea: start with reverse and append operation early on and perform it partially, in order to improve worst case complexity
- rot operation generalizes reverse and append

```
-- rot b e a = b ++ reverse e ++ a, assumes length e = length b + 1
rot :: [a] -> [a] -> [a] -> [a]
rot [] [x] a = x : a
rot (x : b) (y : e) a = x : rot b e (y : a)
```

- observation: with each step of rot, at least one element of resulting list is produced
- improved queue implementation is based on rot, it stores lengths of both lists and keeps invariant: length e <= length b
- improved execution costs: worst case $O(exercise(n))$, amortized cost: $O(1)$

An Improved Implementation

```
data Queue2 a = Queue2 Int [a] Int [a]

empty2 :: Queue2 a
empty2 = Queue2 0 [] 0 []

insert2 :: a -> Queue2 a -> Queue2 a
insert2 x (Queue2 lb b le e) = makeQ2 lb b (le + 1) (x : e)

-- assumes le <= lb + 1
makeQ2 :: Int -> [a] -> Int -> [a] -> Queue2 a
makeQ2 lb b le e
  | le <= lb = Queue2 lb b le e
  | otherwise = Queue2 (lb + le) (rot b e []) 0 []

remove2 :: Queue2 a -> (a, Queue2 a)
remove2 (Queue2 _ [] _ _) = error "empty queue"
remove2 (Queue2 lxb (x : b) le e) = let newQ = makeQ2 (lxb - 1) b le e
 in seq newQ (x, newQ)
```


Worst Case Complexity

- the improved implementation does not have $O(1)$ worst case complexity (see exercise)
- problem: although rot delivers one element per recursion step, there might be nested rot occurrences
- solution: enforce that the rot-list is further evaluated on every insertion and removal operation
- technique: create two shared copies where the second copy is used to trigger evaluation of the spine of the list
- upcoming implementation of Okasaki has worst case complexity of $O(1)$ for each queue operation
- invariants for Queue3 b e b'
  - b' is a sublist of b, used for triggering evaluation of b
  - length e <= length b (as before)
  - length b' = length b - length e

Okasaki's Real Time Implementation of Purely Functional Queues

```
data Queue3 a = Queue3 [a] [a] [a] -- Queue3 b e b', lb' = lb - le, le <= lb

empty3 :: Queue3 a
empty3 = Queue3 [] [] []

insert3 :: a -> Queue3 a -> Queue3 a
insert3 x (Queue3 b e b') = makeQ3 b (x : e) b'

remove3 :: Queue3 a -> (a, Queue3 a)
remove3 (Queue3 [] _ _) = error "empty queue"
remove3 (Queue3 (x : b) e b') = let
   newQ = makeQ3 b e b'
 in seq newQ (x, newQ)

makeQ3 :: [a] -> [a] -> [a] -> Queue3 a
makeQ3 b e (_ : b') = Queue3 b e b'
makeQ3 b e [] = let b' = rot b e [] in Queue3 b' [] b'
```

Example Evaluation using Single Steps for Queue Operations

```
  foldl (flip insert3) empty3 [1..10]
= foldl ... (Queue3 [] [] []) [1..10]
= foldl ... (insert3 1 (Queue3 [] [] [])) [2..10]
= foldl ... (makeQ3 [] [1] []) [2..10]
= foldl ... (Queue3 (rot [] [1] []) [] (rot [] [1] [])) [2..10]
= foldl ... (insert3 2 (Queue3 (rot [] [1] []) [] (rot [] [1] []))) [3..10]
= foldl ... (makeQ3 (rot [] [1] []) [2] (rot [] [1] [])) [3..10]
= foldl ... (makeQ3 [1] [2] [1]) [3..10]
= foldl ... (Queue3 [1] [2] []) [3..10]
= foldl ... (insert3 3 (Queue3 [1] [2] [])) [4..10]
= foldl ... (makeQ3 [1] [3,2] []) [4..10]
= foldl ... (Queue3 (rot [1] [3,2] []) [] (rot [1] [3,2] [])) [4..10]
= foldl ... (insert3 4 (Queue3 (rot [1] [3,2] []) [] (rot [1] [3,2] []))) [5..10]
= foldl ... (makeQ3 (rot [1] [3,2] []) [4] (rot [1] [3,2] [])) [5..10]
= foldl ... (makeQ3 (1 : rot [] [2] [3]) [4] (1 : rot [] [2] [3])) [5..10]
= foldl ... (Queue3 (1 : rot [] [2] [3]) [4] (rot [] [2] [3])) [5..10]
```


Example Evaluation Continued

- from now on only show intermediate steps, but not single steps

```
  foldl (flip insert3) empty3 [1..10]
= ... (Queue3 (1 : rot [] [2] [3]) [4] (rot [] [2] [3])) [5..10]
= ... (Queue3 [1,2,3] [5,4] [3]) [6..10]
= ... (Queue3 [1,2,3] [6,5,4] []) [7..10]
= ... (Queue3 (rot [1,2,3] [7,6,5,4] []) [] (rot [1,2,3] [7,6,5,4] [])) [8..10]
= ... (Queue3 (1 : rot [2,3] [6,5,4] [7]) [8] (rot [2,3] [6,5,4] [7])) [9..10]
= ... (Queue3 (1 : 2 : rot [3] [5,4] [6,7]) [9,8] (rot [3] [5,4] [6,7])) [10]
= ... (Queue3 (1 : 2 : 3 : rot [] [4] [5,6,7]) [10,9,8] (rot [] [4] [5,6,7])) []
= Queue3 (1 : 2 : 3 : rot [] [4] [5,6,7]) [10,9,8] (rot [] [4] [5,6,7])
```


Final Remarks on Purely Functional Queues

- Okasaki's implementation heavily relies upon sharing and lazy evaluation
- using ideas of Okasaki's queue implementation can be used to obtain a worst-case $O(1)$ implementation of double ended queues (deques)
  (with push and pop operations at both ends)
- there are alternative purely functional deque implementations with $O(1)$ worst case behavior that do not depend on lazy evaluation, but have a more complex implementation (Kaplan, Tarjan: Purely functional, real-time deques with catenation)

Exercises

1. Consider the simple queue implementation. The amortized complexity is $O(1)$. This implies that n consecutive operations have cost $O(n)$.
   However, this statement is only true, if always the same queue is used and the queue is not copied. Write a Haskell program that performs $O(n)$ many queue operations (insertion, removal, and queue-copying), and requires $\Theta(n^2)$ time.
   Use `cabal repl Exercise12` and test using `:set +s` within ghci. Reason: Full compilation might optimize and tune your code so that a quadratic behavior in ghci might not be visible after compilation.
2. Study the improved implementation `Queue2`. Perform an evaluation of iterated insertion in the style of Slides 25 and 26 for `Queue2` to identify a pattern in the evaluation. Afterwards derive a lower bound on the worst case complexity of `remove2` after a sequence of n many insertions.
3. Improve the implementation of the LPO-encoding. Via profiling it was figured out that the lookup via term keys is expensive. To this end, introduce term indices for the lookup, and use mutable arrays of type `STArray` for storing the memoized results.
   Perform profiling before and after your modification and briefly report on the results.

Literature

- Real World Haskell, Chapter 25
- https://downloads.haskell.org/ghc/latest/docs/users_guide/profiling.html
- https://hackage.haskell.org/package/base/docs/Control-Monad-ST.html
- https://hackage.haskell.org/package/base/docs/Data-STRef.html
- https://hackage.haskell.org/package/array/docs/Data-Array-ST.html
- https://hackage.haskell.org/package/array/docs/Data-Array-MArray.html
- Chris Okasaki, Simple and Efficient Purely Functional Queues and Deques. J. Funct. Program. 5(4): 583-592 (1995)