



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 `MVars`
- 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

RT (DCS @ UIBK)

Week 12

2/25

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

COST CENTRE		SRC	entries	%time	%alloc
\$f0rdList_\$\$ccompare1	libraries/ghc-prim/GHC/Classes.hs:486:5-11			23.9	0.0
\$f0rdTerm_\$\$ccompare	src/TRS.hs:6:49-51			22.0	0.0
\$f0rdList_\$\$ccompare	libraries/ghc-prim/GHC/Classes.hs:486:5-11			21.6	0.0
compare	libraries/ghc-prim/GHC/Classes.hs:424:5-46			19.5	0.0
COST CENTRE		SRC	entries	%time	%alloc
MAIN	<built-in>		0	100.0	100.0
main	Exercise11.hs:127:1-4		1	99.6	99.9
lpoSolver1	<no location info>		566	47.5	48.0
runSmtSolver1	<no location info>		566	47.5	48.0
\$wrunSmtEncoder	Abstract_SMT_Encoder.hs:32:1-13		566	47.2	47.8
\$wlpoEncoder	<no location info>		551237	45.2	10.2
\$f0rdTerm_\$\$ccompare	TRS.hs:6:49-51		342219684	28.8	0.0
\$f0rdTerm_\$\$ccompare	TRS.hs:6:49-51		174072475	14.8	0.0
reverseLpoSolver1	<no location info>		566	47.3	48.0
parseAri	Parser_ARI.hs:39:1-8		566	0.3	3.2

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 readability
 - there are (deleted) time/alloc columns for costs that are caused by the current cost center
 - the displayed time/alloc columns are for the accumulated costs
- important: **external costs do not occur in the data**, e.g., costs 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]`

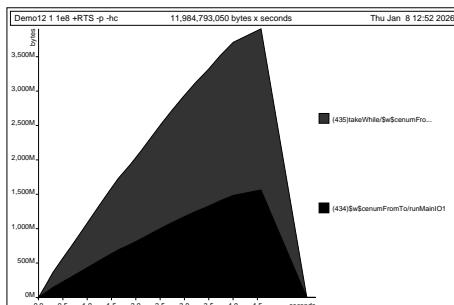
```
main = ... -- read d via getArgs and print "mean [1..d]"
mean1 :: [Double] -> Double
mean1 xs = ({-# SCC "sum" #-} sum xs)
  / fromIntegral ({-# SCC "len" #-} length xs)
```
- running the demo (1 = select mean1 function, 1e8 is value of d) with statistics


```
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 -l`

```
cabal run Demo12 --enable-profiling --profiling-detail none
  -- 1 1e8 +RTS -p -hc -l
```
- inspect generated data of files `Demo12.hp` and `Demo2.eventlog` via
 - old: `hp2ps Demo12.hp && ps2pdf Demo12.ps && open Demo12.pdf`
 - new: `eventlog2html Demo12.eventlog -o Demo12_ev1.html`
(installation via: `cabal install eventlog2html`)

Demo12.hp – mean1

- obtain graph



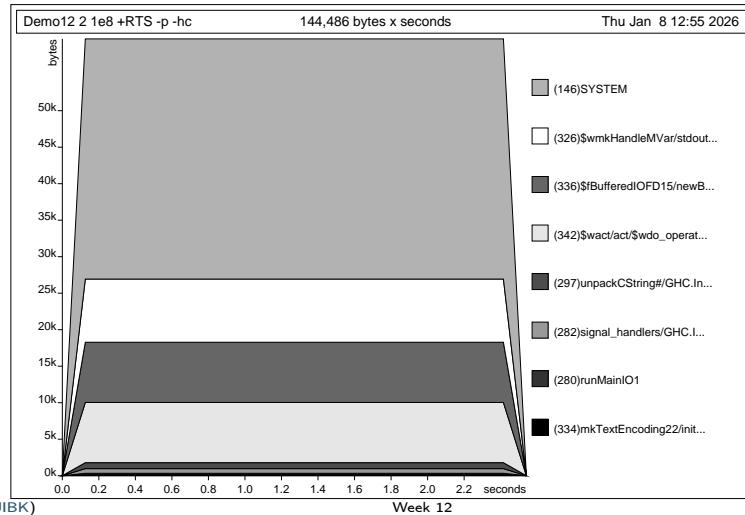
- code analysis
 - generated list `[1..d]` is completely constructed in memory
 - problem: list elements are generated one-by-one for summation, and list needs to be kept for computing its length
 - interpretation of graph: once the length computation starts, memory can be freed again

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., $0 + x_1 + x_2 + x_3 + \dots$
- both `+RTS -s` and `+RTS -p -hc -l` confirm improved memory usage (runtime is improved, too: less time required for garbage collection)

Demo12.hp – mean2



RT (DCS @ UIBK)

Week 12

Thu Jan 8 12:55 2026

13/25

RT (DCS @ UIBK)

Week 12

14/25

Efficient Data Structures

Choice of Data Structures

- efficiency is often obtained by choosing suitable data structures
- we have already seen one possibility
 - use **mutable data structures** within **purely functional code** via **ST-monad**, **STRef-** and **STArray-types**
- upcoming: an example of a **data structure designed for pure functional programming**

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

RT (DCS @ UIBK)

Week 12

15/25

RT (DCS @ UIBK)

Week 12

16/25

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
 - $\text{rot } b \ e \ a = b \ ++ \ \text{reverse } e \ ++ \ a$, assumes $\text{length } e = \text{length } b + 1$
 - 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** \leq **length b**
- improved execution costs: worst case $O(\text{exercise}(n))$, 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 algorithm 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** \leq **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)

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)