Functional Programming and Verification

Prof. Dr. H. Seidl, N. Hartmann, R. Vogler WS 2018/19

Exercise Sheet 6 Deadline: 02.12.2018

General Information

Detailed information about the lecture, tutorials and homework assignments can be found on the lecture website[1]. Solutions have to be submitted to Moodle[2]. Upload your solutions as a single file named 'hw06.ml' to moodle. Since submissions are tested automatically, solutions that do not compile or do not terminate within a given time frame cannot be graded and thus result in 0 Points. If you do not manage to get all your stuff compiling and/or terminating, comment the corresponding parts of the file! Use Piazza[3] to ask questions and discuss with your fellow students.

Functional Programming

Since this is a course about functional programming, we restrict ourselves to the functional features of OCaml. In other words: The imperative and object oriented subset of OCaml must not be used.

Assignment 6.5 (H) Peano Arithmetic [3 Points]

The natural numbers can be defined recursively as follows:

- 0 is a natural number.
- if n is a natural number, then so is the successor of n

We can easily represent this in OCaml using corresponding constructors:

```
type nat = Zero | Succ of nat
```

Implement the following functions for natural numbers:

1. `int_to_nat : int -> nat` converts an integer to natural.
2. `nat_to_int : nat -> int` converts a natural to integer.
3. `add : nat -> nat -> nat` adds two natural numbers.
4. `mul : nat -> nat -> nat` multiplies two natural numbers.
5. `pow : nat -> nat -> nat` a call `pow a b` computes a^b.
6. `leq : nat -> nat -> bool` a call `leq a b` computes $a \leq b$.

You are **not** allowed to use the first two functions to implement the rest!

[1] https://www.in.tum.de/i02/lehre/wintersemester-1819/vorlesungen/functional-programming-and-verification/

[2] https://www.moodle.tum.de/course/view.php?id=44932

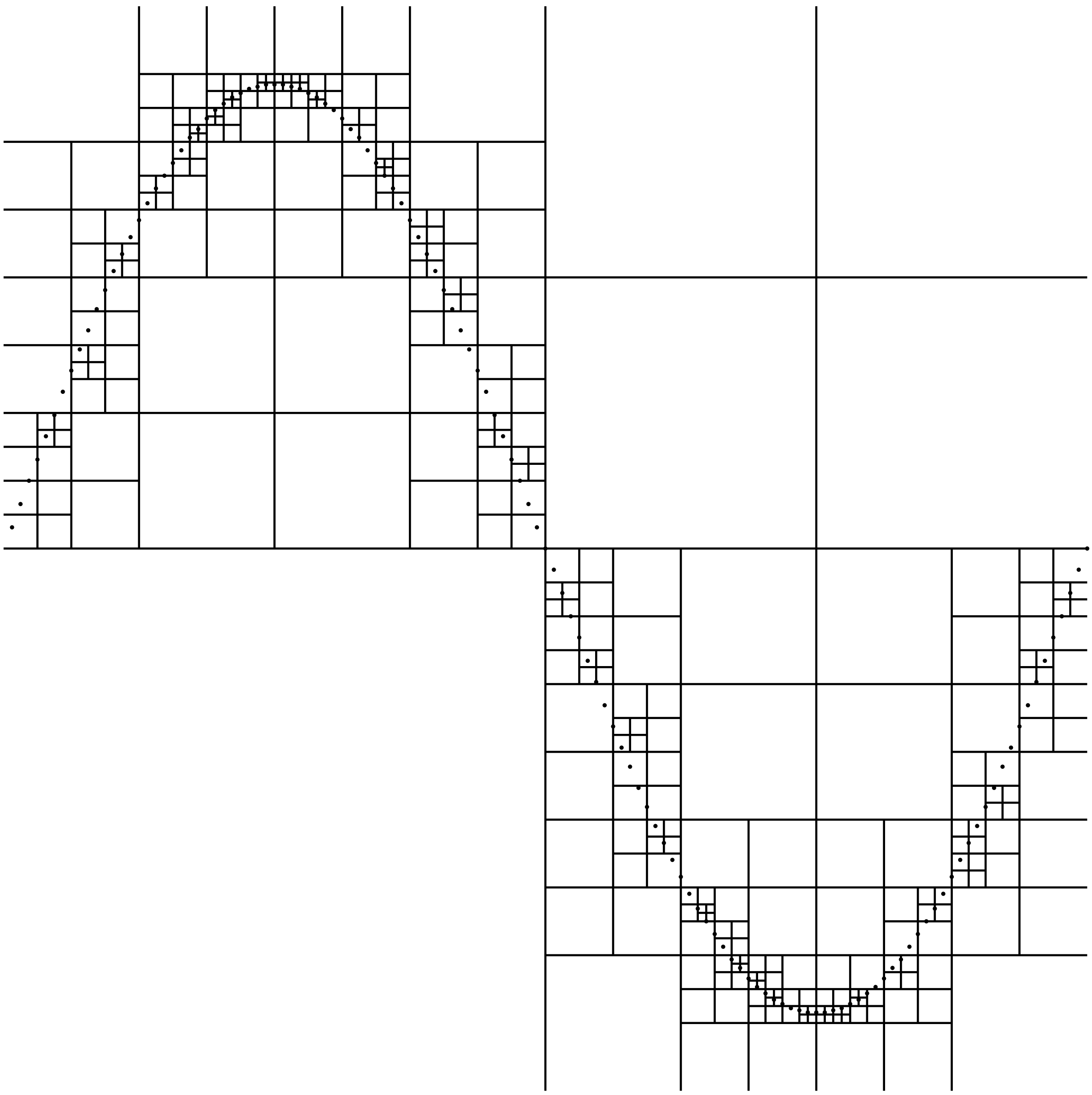
[3] https://piazza.com/tum.de/fall2018/in0003/home

Assignment 6.6 (H) Quadtrees [6 Points]

The quadtree data structure stores points by recursively partitioning a 2-dimensional region.

Let $Rect(x_1, y_1, x_2, y_2)$ denote the rectangular region with lower left corner (x_1, y_1) and upper right corner (x_2, y_2). We define that all points (x, y) with $x_1 \leq x < x_2$ and $y_1 \leq y < y_2$ belong to this region. Every internal node represents a rectangular region $Rect(x_1, y_1, x_2, y_2)$ and has exactly four children, which represent the equally sized subregions $Rect(x_1, y_1, x_c, y_c), Rect(x_1, y_c, x_c, y_2), Rect(x_c, y_1, x_2, y_c)$ and $Rect(x_c, y_c, x_2, y_2)$, respectively, where $(x_c, y_c) = (\frac{x_2-x_1}{2}, \frac{y_2-y_1}{2})$ is the nodes center point. Leafs are either empty or contain a single point.

At all times, every subtree rooted at node N has the minimal height required to store all points that are inside the region corresponding to N. Every point (x, y) is stored at most once, so if the same point is inserted a second time, the tree is unchanged. The following figure shows a quadtree of size 512x512 that stores 100 points along the sine wave:



We represent a quadtree node by the following sum type:

```
type quadtree_node = NoPoint
                   | Point of int * int
                   | QNode of quadtree_node (* bottom left *)
                            * quadtree_node (* top left *)
                            * quadtree_node (* bottom right *)
                            * quadtree_node (* top right *)
```

and the entire quadtree is then represented by the `quadtree` type:

```
type quadtree = { width:int; height:int; root:quadtree_node }
```

Note, that we do not store the region inside the individual nodes, but just in the tree itself, so node regions must be computed during tree traversal.

Implement the function `insert : point -> quadtree -> quadtree` that inserts a point into the tree.

Hint: You may assume that region sizes are always a power of 2.

Hint: The function `print_quadtree` *is provided to save an svg-image of your tree.*

Assignment 6.7 (H) Expression Evaluation [4 Points]

We define expressions over rationals using these OCaml types:

```
type rat = int * int (* num, denom *)
type unary_op = Neg
type binary_op = Add | Sub | Mul | Div
type expr = Const of rat
          | UnOp of unary_op * expr
          | BinOp of binary_op * expr * expr
```

Implement a function `eval_expr : expr -> rat` that evaluates the given expression. The resulting fraction need not be simplified, so `(-3,2)`, `(3,-2)`, `(-6,4)`, `(12,-8)`, ... are all accepted. Example:

```
eval_expr (BinOp (Mul, BinOp (Sub, Const (3,8), Const (2,4)), Const (6,-3)))
```

evaluates to `(1,4)` (or `(2,8), ...`).

Assignment 6.8 (H) Crawling on Trees [7 Points]

Once again, consider binary trees, which we define as:

```
type tree = Empty | Node of int * tree * tree
```

In this assignment you are supposted to implement a crawler that walks along binary trees and performs different operations. At any time, the crawler "sits" on a particular node of the tree (this includes the `Empty`-leaf). In the following we refer to this node as the *current node*. Furthermore, the crawler uses a stack to store trees. Initially, the crawler is positioned at the input tree's root and is then instructed using the commands

```
type command = Left | Right | Up | New of int | Delete | Push | Pop
```

with the following meaning:

- `Left` moves the crawler to the current node's left child.
- `Right` moves the crawler to the current node's right child.
- `Up` moves the crawler up to the current node's parent node.
- `New x` replaces the current node (including all children) with a new node with value `x`.
- `Delete` removes the current node (including all children) leaving behind an `Empty`-leaf.
- `Push` pushes the subtree rooted at the current node onto the stack. The tree stays unchanged.
- `Pop` replaces the subtree rooted at the current node with the topmost tree of the stack. The tree is then popped from the stack.

Implement a function `crawl : command list -> tree -> tree` that executes a list of crawler commands on the given tree. You may assume that the list of commands is always valid, so there is no `Left` or `Right` when the crawler is already at a leaf, no `Up` when it is on the root and no `Pop` when the stack is empty.

Hint: The tricky part is to get the `Up` *command right. If you do not manage to implement this correctly, leave it out and you will still get some points for the rest.*

Hint: The function `print_tree` *is provided, that can be used to dot[4]-export your tree.*

[4] `https://en.wikipedia.org/wiki/DOT_(graph_description_language)`