Functional Programming and Verification

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

Exercise Sheet 7 — Deadline: 09.12.2018

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 'hw07.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 7.4 (H) Fun with Folding [7 Points]

Find a function `f`n, such that

1. `fold_left f1 0 l` returns the length of list `l`.
2. `fold_left f2 []` $[l_1; l_2; \ldots; l_n]$, for l_i being arbitrary lists, returns the longest of those lists. If multiple lists have maximal length, either one may be returned.
3. `fold_left f3 []` $[a_1, b_1; a_2, b_2; \ldots; a_n, b_n]$ for arbitrary a_i, b_i computes $[b_1, a_1; b_2, a_2; \ldots; b_n, a_n]$.
4. `fold_left f4 []` $[a_0; \ldots; a_{n-3}; a_{n-2}; a_{n-1}; a_n]$ for arbitrary elements a_i computes $[a_n; a_{n-2}; \ldots; a_0; \ldots; a_{n-3}; a_{n-1}]$.
5. `fold_left f5 (fun _ -> 0)` $[k_1, v_1; k_2, v_2; \ldots; k_n, v_n]$ computes a function g such that $g(k_i) = v_i$ for all $1 \leq i \leq n$. Assume that $\forall_{1 \leq i < j \leq n}. k_i \neq k_j$.
6. `fold_left f6 [0]` $[f_1; f_2; \ldots; f_n]$ computes $[f_n(\cdots f_2(f_1(0)) \cdots); \ldots; f_2(f_1(0)); f_1(0); 0]$ for unary functions f_i.
7. `fold_left f7` a $[c_n; c_{n-1}; \ldots; c_0]$ computes $a^{2^{n+1}} * \prod_{i=0}^{n} c_i^{2^i}$ for integers $a, c_0, \ldots c_n$.

---

[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 7.5 (H) Expression Evaluation [7 Points]

In this assignment you are supposed to extend the expression evaluation of assignment 6.7. Therefore, we extend our expression language as follows:

```
type rat = int * int (* num, denom *)
type var = string (* new *)
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
          | Var of var (* new *)
          | Func of var * expr (* new *)
          | Bind of var * expr * expr (* new *)
          | App of expr * expr (* new *)
          | Ite of expr * expr * expr (* new *)
type value = Rat of rat | Fun of var * state * expr (* new *)
and state = var -> value option (* new *)
```

First, we introduce 5 new forms of expressions:

- `Var x` simply represents a variable with name `x` in an expression. Example: An expression $i + \frac{1}{3}$ is now represented as

  ```
  BinOp (Add, Var "i", Const (1,3))
  ```

- `Ite (c, t, e)` represents an expression `if c then t else e`, where `c` is the condition and `t` and `e` are the expressions at the *then*- and *else*-branch, respectively. Note, that our language does not contain a `bool` type, so we consider a condition as `false` if and only if the value `c` is equal to 0 (e.g. $\frac{0}{3}, \frac{0}{8}, \ldots$).

- `Bind (x, e, b)` binds the value of expression `e` to the variable named `x` inside the expression `b`. All bindings are non-recursive. Example: An expression like `let t = 2/3 * i in t` can be represented by

  ```
  Bind ("t", BinOp (Mul, Const (2, 3), Var "i"), Var "t")
  ```

- `Func (a, b)` is the definition of a unary function with argument `a` and function body `b`. Example: The expression `fun y -> 2 - y` is represented by

  ```
  Func ("y", BinOp (Sub, Const (2,1), Var "y"))
  ```

- `App (f, a)` is the application of the function produced by expression `f` to argument `a`. Note that both `f` and `a` may be arbitrarily complex. Example: `(foo i) (1/2 - 2/7)` is represented by

  ```
  App (App (Var "foo", Var "i"), BinOp (Sub, Const (1,2), Const (2,7)))
  ```

Next, in addition to rationals, expressions can now also evaluate to functions. Therefore we define the `value` type where `Rat` represents a rational (as before) and `Fun (a, s, b)` represents a function with argument `a`, body `b` and captured state `s`. Like in OCaml, the values of variables inside a function (except for the function's arguments) are determined

when the function is defined, not when it is called. Thus, a function value needs to store the state of variables in `s`.

Finally, we define such a `state` as a mapping from variable names to `value option`, such that the value of a variable is `None`, if is has not been defined yet.

Extend the function `eval_expr : state -> expr -> value` to support these new features. Note, that the type of `eval_expr` is adapted accordingly: A `state`, which is the current mapping of variables to values, is passed as the first argument and `eval_expr` now returns a `value`, since an expression may no longer only evaluate to a rational.

Hint: The file 'hw07.ml' already contains an adapted version of the ***`eval_expr`*** *function handling the expressions introduced in assignment 6.7.*

Hint: In this new scenario, type conflicts may rise, e.g. if the first expression of a function application does not evaluate to a function. These cases need not be handled in a particular way, you can just call ***`failwith`*** *`"invalid type"`.*

Assignment 7.6 (H) Minimal Spanning Tree [6 Points]

We define a weighted undirected graph by the list of its edges:

```
type graph = (int * float * int) list
```

Thus, `[(0,1.5,1); (0,2.5,2)]` represents the graph containing nodes $0, 1, 2$ and edges with weight 1.5 and 2.5 between node 0 and nodes 1 and 2, respectively.

Implement a function `mst : graph -> graph` that computes the minimum spanning tree for the given graph. You may freely choose your algorihm. Note, that the function returns the subgraph that forms the minimum spanning tree. The edges may be output in any order.

Hint: The ***`List`*** *module provides a lot of useful functions for this assignment!*

Hint: The solution uses Prim's algorithm[4]*, however, you may choose your favourite algorithm.*

[4] `https://en.wikipedia.org/wiki/Prim%27s_algorithm`