Programming Paradigms Fall 2023
Homework Assignment №3

Innopolis University

November 30, 2023

About this assignment

You are encouraged to use homework as a playground: don't just make some code that solves the problem, but instead try to produce a readable, concise and well-documented solution.

Try to divide a problem until you arrive at simple tasks. Create small functions for every small logical task and combine those functions to build a complex solution. Try not to repeat yourself: for every logical task try to have just one small function and reuse it in multiple places.

The assignment is split into mutliple exercises that have complexity specified in terms of stars ($\star$). The more stars an exercise has the more difficult it is. Exercises with three or more stars ($\star\star\star$) might be really challenging, so please make sure you are done with simple exercises before trying out the more difficult ones.

The assignment provides clear instructions and some examples. However, you are welcome to make the result extra pretty or add more functionality as long as it does not change significantly the original problem and does not make the code *too complex*.

Submit a homework with all solutions as a single file. The file should work by loading it in DrRacket development environment and simply running it.

This homework will be graded out of 100 point. This homework also contains an extra credit exercise.

1 Binary trees

Consider the following representation of binary trees:

- `'()` — an empty binary tree;
- `` `(node ,value ,left ,right) `` — a node with a value `value` and two subtrees (`left` and `right`).

Exercise 1.1 (⋆, 5 points)**.** Implement a relational program `treeo` that checks whether a given term is a valid tree.

```
(run* (q) (treeo 'empty))
; '(_.0)

(run* (q) (treeo '(node 1 empty (node 2 (node 3 empty empty) empty))))
; '(_.0)

(run 5 (q) (treeo q))
; '(empty
;   (node _.0 empty empty)
;   (node _.0 empty (node _.1 empty empty))
;   (node _.0 (node _.1 empty empty) empty)
;   (node _.0 empty (node _.1 empty (node _.2 empty empty))))
```

Exercise 1.2 (⋆⋆, 5 points)**.** Implement predicate `prefix-subtreeo` such that `(prefix-subtreeo small large)` is `#t` when `small` is contained in `large`:

1. an empty tree is contained in any tree;
2. a non-empty tree is contained in another non-empty tree when they have the same root and both subtrees are contained in the other subtrees respectively;

```
(run* (q) (prefix-subtreeo q '(node 3 (node 1 empty (node 2 empty empty)) (node 4 empty empty))))
; '(empty
;   (node 3 empty empty)
;   (node 3 empty (node 4 empty empty))
;   (node 3 (node 1 empty empty) empty)
;   (node 3 (node 1 empty empty) (node 4 empty empty))
;   (node 3 (node 1 empty (node 2 empty empty)) empty)
;   (node 3 (node 1 empty (node 2 empty empty)) (node 4 empty empty)))
```

Exercise 1.3 (⋆⋆, 10 points)**.** Implement predicate `preordero` such that `(preorder tree vals)` is `#t` when `vals` contains exactly of values from `tree` in **preorder traversal** order.

```
(run* (q) (preordero
  '(node 1 (node 2 (node 3 empty empty) empty) (node 4 empty empty))
  '(3 2 1 4)))
; '()

(run* (q) (preordero
  '(node 1 (node 2 (node 3 empty empty) empty) (node 4 empty empty))
  q))
; '((1 2 3 4))

(run* (q) (preordero
  q
  '(1 2 3)))
; '((node 1 empty (node 2 empty (node 3 empty empty)))
;   (node 1 empty (node 2 (node 3 empty empty) empty))
;   (node 1 (node 2 empty empty) (node 3 empty empty))
;   (node 1 (node 2 empty (node 3 empty empty)) empty)
;   (node 1 (node 2 (node 3 empty empty) empty) empty))

(run 7 (q) (fresh (xs) (preordero q xs)))
; '(empty
;   (node _.0 empty empty)
;   (node _.0 empty (node _.1 empty empty))
;   (node _.0 (node _.1 empty empty) empty)
;   (node _.0 empty (node _.1 empty (node _.2 empty empty)))
;   (node _.0 (node _.1 empty empty) (node _.2 empty empty))
;   (node _.0 empty (node _.1 (node _.2 empty empty) empty)))
```

2 Simple expressions

Exercise 2.1 (⋆, 5 points). Implement predicate `expro` that checks if a given term is a valid arithmetic expression:

1. a number;
2. a term `(+ X Y)` where both `X` and `Y` are valid expressions;
3. a term `(* X Y)` where both `X` and `Y` are valid expressions;

```
(run* (e) (expro '(+ 2 (* 3 4))))
; '(_.0)

(run* (e) (expro '(+ 2 (* e 4))))
; '()
```

Exercise 2.2 (⋆⋆, 10 points). Implement predicate `expro-with` such that `(expro-with e subexprs)` is `#t` when `e` is an expression term that uses each subexpression from `subexprs` once (in that order). You may assume that `subexprs` has known shape (i.e. length):

```
(run* (e) (expro-with e '(1 2)))
; '((+ 1 2) (* 1 2))

(run* (e) (expro-with e '(1 2 3)))
; '((+ 1 (+ 2 3))
;   (+ (+ 1 2) 3)
;   (+ 1 (* 2 3))
;   (+ (* 1 2) 3)
;   (* 1 (+ 2 3))
;   (* (+ 1 2) 3)
;   (* 1 (* 2 3))
;   (* (* 1 2) 3))
```

Exercise 2.3 (⋆, 5 points). Implement a function `equations-with` such that `(equations-with subexprs)` returns a list of equations in the form `(= val expr)` where `expr` is an expression generated from subexpressions `subexpr` and `val` is its corresponding numeric value.

```
(equations-with '(1 2 3))
; '((= 6 (+ 1 (+ 2 3)))
;   (= 6 (+ (+ 1 2) 3))
;   (= 7 (+ 1 (* 2 3)))
;   (= 5 (+ (* 1 2) 3))
;   (= 5 (* 1 (+ 2 3)))
;   (= 9 (* (+ 1 2) 3))
;   (= 6 (* 1 (* 2 3)))
;   (= 6 (* (* 1 2) 3)))
```

Exercise 2.4 (⋆, 5 points)**.** Implement a relational program `atomic-expro` that checks that a given expression is atomic (not a binary operation expression).

```
(run* (q) (fresh (x) (atomic-expro 2)))
; '(_.0)
(run* (q) (fresh (x) (atomic-expro x)))
; '(_.0)
(run* (q) (fresh (x) (atomic-expro `(+ 1 ,x))))
; '()
```

Exercise 2.5 (⋆, 5 points)**.** Implement a relational program `root-equiv-expro` that checks if two expressions are related by one of the following equations directly:

1. $x + y = y + x$ for all x, y
2. $x + 0 = x$ for all x
3. $x \times 1 = x$ for all x
4. $x \times 0 = 0$ for all x
5. $(x \times y) \times z = x \times (y \times z)$ for all x, y, z
6. $x \times (y \times z) = (x \times y) \times z$ for all x, y, z
7. $(x + y) + z = x + (y + z)$ for all x, y, z
8. $x + (y + z) = (x + y) + z$ for all x, y, z
9. $(x + y) \times z = (x \times z) + (y \times z)$ for all x, y, z
10. $x \times (y + z) = (x \times y) + (x \times z)$ for all x, y, z

```
(run* (e) (root-equiv-expro '(+ x (* 0 y)) e))
; '((+ (* 0 y) x))
(run* (e) (root-equiv-expro '(* x (* 0 y)) e))
; '((* (* 0 y) x) (* (* x 0) y))
(run* (e) (root-equiv-expro '(* 1 (* 0 y)) e))
; '((* 0 y) (* (* 0 y) 1) (* (* 1 0) y))
```

Exercise 2.6 (⋆, 5 points)**.** Implement a relational program `equiv-expro` that checks if two expressions are related by one of the equations above, possibly somewhere in a subterm:

```
(run* (e) (equiv-expro '(+ x (* 0 y)) e))
; '((+ (* 0 y) x) (+ x 0) (+ x (* y 0)))
(run* (e) (equiv-expro '(* x (* 0 y)) e))
; '((* (* 0 y) x) (* (* x 0) y) (* x 0) (* x (* y 0)))
(run* (e) (equiv-expro '(* 1 (* 0 y)) e))
; '((* 0 y) (* (* 0 y) 1) (* (* 1 0) y) (* 1 0) (* 1 (* y 0)))
```

Exercise 2.7 (⋆, +1% extra credit)**.** Implement a relational program `equiv-expr-patho` that checks (and provides a corresponding equational chain) if two expressions are related by zero or more applications of the equations above, possibly somewhere in the subterms:

```
(run 1 (path) (fresh (e) (equiv-expr-patho
  '(* (* 1 (* z x)) (+ 0 y))
  '(* y (* x z))
  path)))
; '(((* (* 1 (* z x)) (+ 0 y))
;     (* (* 1 (* z x)) y)
;     (* (* z x) y)
;     (* (* x z) y)
;    (* y (* x z))))

(run 1 (path) (fresh (e) (equiv-expr-patho
  '(+ (* (+ 1 0) (+ x 0)) (* 0 y))
  '(+ x 0)
  path)))
; '(((+ (* (+ 1 0) (+ x 0)) (* 0 y))
;    (+ (* (+ 1 0) (+ x 0)) 0)
;    (+ 0 (* (+ 1 0) (+ x 0)))
;    (* (+ 1 0) (+ x 0))
;    (+ (* 1 (+ x 0)) (* 0 (+ x 0)))
;    (+ (* 0 (+ x 0)) (* 1 (+ x 0)))
;    (+ 0 (* 1 (+ x 0)))
;    (* 1 (+ x 0))
;    (+ x 0)))
```

3 Custom implementation of μKanren

For each of the exercises 1.1–1.3 and 2.1–2.3, you can receive extra credit (+0.1% extra credit per point), if you implement your own version of μKanren and solve these exercises using your own implementation. You can use any language for the implementation, except Haskell and any language from the LISP family, unless your implementation significantly differs from the corresponding lecture versions.

To receive the extra credit, you might be required to defend your implementation in person, during a special session or during the oral exam (at the latest).