Projects for Advanced Functional Programming 2017

Deadline: 12th January 2018, 23:59

General

- For this part of the course, you are encouraged to work in **groups of two** in pair programming[1] style.
- Each group has to provide working solutions **for two out of the three** following projects.
- **All** group members should be able to explain the design and implementation for **both** projects.
- Each group can get **up to 20 points for each project (40 points in total)**:
  - 10 points are awarded for providing a working solution that satisfies the **"Requirements"** and
  - extra points are awarded for satisfying the **"Extra requirements"** of each project.
- For each project, in addition to your code, you should also provide a {`sat, sql, tic`}`-report.pdf` document with a brief but comprehensive description of your solution (include in this description a short paragraph on who did what in your team — this will not affect your individual grades) and **any extra material** that you used to check correctness and performance of your program (for example, any tests or benchmarks you created in the process).
- Make sure that you **mention in your report any extra requirements you fulfilled**.
- You can **register your teams** on Studentportalen. Each team should upload an `afp_project.zip` in the relevant section, including at least the requested programs and the relevant reports. Any additional material you include should be referenced in the reports.
- Students who want to do this part of the course alone are **still** required to submit two out of the three projects. No special rules apply.

Have fun!

1 3-SAT server

`sat_server.erl`, **Erlang**

Building upon the experience you got from Assignment 1, you are now asked to implement a server that will solve instances of the 3-SAT problem. 3-SAT is the simplest NP-complete problem, and is defined in the following way: Given a boolean formula in Conjunctive Normal Form, where each clause has at most 3 variables, find an assignment for the variables that makes the formula evaluate to `true`. You can read more about 3-SAT on a good algorithms' book or on Wikipedia.

For this project, instances of 3-SAT will be encoded as lists of 3-tuples, each element being a (non-zero) integer. Negative integers mark negation of the respective variable. For example, the formula: $(var_3 \vee \neg var_4 \vee var_7) \wedge (var_1 \vee var_2 \vee \neg var_3)$ is encoded with the following Erlang term: `[{3,-4,7},{1,2,-3}]`.

Requirements

- The server must be started with a call to `sat_server:start_link()`.
- The server must listen for *multiple connections* on port `3547`.

[1] https://en.wikipedia.org/wiki/Pair_programming

- The server must reply within 1 second with a `ready` message or a `busy` message after a connection is made. If the reply is `busy`, the server must then close the connection.
- The server may reply with `busy` *only* if it has already 8 or more active connections.
- After sending the `ready` message, the server must wait to receive a 3-SAT instance encoded in the format specified above. No dot should be expected in the end of the instance.
- After receiving an instance, the server must immediately send back a `trying` message.
- The server should then send a new reply within 10 seconds. The reply can be:
  - `{sat,[Var1,Var2,...]}`, where `Var1, Var2, ...` are boolean values (`true` or `false`) that make the formula satisfiable when assigned to the respective variables.
  - `unsat`, if the formula is not satisfiable.
  - `aborted`, if the server decides to not find a solution for the formula.
  - `trying`, if the server is still trying to solve the formula. In this case, another reply is expected within 10 seconds. The new reply can also be `trying`.
- The server must solve formulas with up to 20 variables within 10 seconds. Some test cases are provided in **instances.txt**[2], in which, a list of tuples (expected answer and input) are defined.
- There is no limit on how many variables a formula can have (the server may abort on a huge one).
- The server must not crash if a connection is terminated unexpectedly.
- The server must reply to any unexpected input with the message `ignored` within 1 second. Unexpected input is malformed 3-SAT instances or any message that arrives while still evaluating another 3-SAT instance.

Extra requirements

- Provide (a significant amount of) tests for your solution. (**2 points**)
- Support instances that span over multiple lines. (**1 point**)
- After receiving a `ready` message or a final reply for a previous request, the client is expected to send a (new) 3-SAT instance within 1 minute. Otherwise the server sends a `timeout` message and closes the connection. (**1 point**)
- Stop searching for a solution if a connection is dropped. (**1 point**)
- The server should send a new message every 10 seconds. This message can be `trying`, if the server is still trying to solve the formula, or any of the final messages if it is done. (**1 point**)
- The server interprets an `abort` message as a request to stop trying to solve the previous instance. It should reply with an `aborted` message within 1 second. (**1 point**)
- The server uses multiple workers and a parallel strategy to solve each 3-SAT instance. (**3 points**)

[2] http://www.it.uu.se/edu/course/homepage/avfunpro/ht17/instances.txt

Example

Similar to what's done in Assignment 1, starting the server from the Erlang shell:

```
1> sat_server:start_link().
{ok,<0.35.0>}
```

After the server is started:

```
$ telnet localhost 3547
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
ready
[{3,-4,7},{1,2,-3}]
trying
{sat, [true,true,false,false,true,true,true]}
[{1,1,1},{-1,-1,-1},{2,2,2},{-2,-2,-2},{3,3,3},{-3,-3,-3},...
 %% omitting similar pairs for variables 4-20 %%%
 ...,{21,21,21},{-21,-21,-21}]
trying
trying
unsat
[{this,is,not,an,instance}]
ignored
```

2 SQL-like relational algebra

sql_relations.rkt, **Racket**

Building upon the experience you got from the relations assignment, you are now asked to define a more expressive language for manipulating relations. This language resembles the Structured Query Language (SQL). It has more operators which work on **mutable relations**. The relations expressed in this language also have type specifications for their columns. Relations are still considered to be sets, so each row should appear only once, but no order is specified. **Extra requirements** are marked as such within the definitions.

Definitions

1. (CREATE <relname> ((<colname> <coltype>) (<colname> <coltype>) ...))
   Creates a new relation named <relname>. The colnames are the identifiers for the columns and the coltypes are their respective types. Types can be integer or string.

2. (INSERT INTO <relname> VALUES (<v_a1> <v_a2> ...) (<v_b1> <v_b2> ...) ...)
   Inserts rows into the relation <relname>. Each row must have a value for every column of the relation and each value must have the correct type for the specified column. Erroneous input should be handled appropriately.
   Extra requirement (**3 points**): The INSERT command should be *atomic*: if a row contains an error (i.e., some value is missing or has the wrong type), no rows should be added into the relation.

3.
```
(SELECT (<p_col1> <p_col2> ...)
   FROM (<relname1> <relname2> ...)
  WHERE (<expr1a> <op> <expr1b>)
   <AND/OR> (<expr2a> <op> <expr2b>)
  ...)
```
   This construct combines and extends the project and restrict operators. It returns a list of rows (without column headers), according to the following rules (each part is explained in the order it affects the result):
   - <relname1>, <relname2>, ... are relations. If more than one relations are specified, the "natural join" of the relations must be performed in the order that they appear. This means that first the natural join of the first two relations is performed, followed by the natural join of the resulting relation with the third relation, and so on.
     In a natural join, the result is a relation that contains the columns of the first relation, followed by the columns of the second relation, in the original order. If, however, the relations have columns with the same identifier, they are expected to have the same type and then they appear only once, in the first relation.
     A row is retained in the result of a join only if for each matching column there is another row in the second relation with the same values. If there are multiple such rows, multiple rows are added in the join. If there are no matching columns, all combinations of rows of the first relation with rows of the second relation must exist in the result.
   - WHERE... is **optional** and defines restriction expressions. Only rows that satisfy the restriction are to be retained in the result. The components are the following:
     expr: It can be a literal value or an identifier, which signifies the value that the row under consideration has on the respective column.
     Extra requirement (**4 points**): It may also be a call to an arbitrary Racket function. The arguments to the function can then again be literal values, column identifiers (that have to be replaced with the respective value before the function is called) or deeper Racket expressions.
     op: >,<,=,<=,>=,!=. Notice that this time the operators appear in a bare form, without the . <op> . syntax. >,<,<=,>= only work with integer, while =,!= work with both integer and string.
     <AND/OR>: express conjunction or disjunction with more constraints.

Parentheses are expected to be present when they need to disambiguate the associativity of conjunction and disjunction. Parentheses may appear around an expr *only* if the expression is applying a Racket function (see the extra requirement).

- <p_col1>, <p_col2> are projection columns. Only the values from these columns should be preserved in the final result. Instead of a list of fields, an asterisk (*) can be used to designate that all columns should be returned.

  Extra requirement (**3 points**): At most one of the columns, which must have integer type, can be wrapped in an aggregating function. In this case only the data from one row must be returned (including the aggregated column):
  - MAX(column): the row with the maximum value in the respective column.
  - MED(column): the row with the median value in the respective column. (For even number of rows, the smaller middle value is used.)
  - MIN(column): the row with the minimum value in the respective column.

4. (INSERT INTO <relname> SELECT ...)
   Combines the INSERT INTO and SELECT operations. Inserts the result of a select operation as values into the specified relation.

5. (DELETE FROM <relname> WHERE ...)
   Removes the rows that match the WHERE clause from the specified relation. The syntax of the WHERE clause is the same as in SELECT, but this time it is **mandatory**.

6. (EXPORT <relname> <filename>)
   Exports the data of the specified relation as a CSV file. Each row should be printed on a separate line and columns should be separated with a comma. Strings should be quoted and quotes within strings should be escaped with a '\' character.

Example

```
$ racket
Welcome to Racket v6.10.1.
> (require "sql_relations.rkt")
> (define (yearly_salary m) (* 12 m))
> (CREATE departments ((department integer) (dept_name string)))
> (INSERT INTO departments VALUES
    (1 "Sales")
    (2 "Marketing"))
> (SELECT * FROM (departments))
'((1 "Sales") (2 "Marketing"))
> (CREATE employees
     ((emp_name string) (department integer) (salary integer) (hired integer)))
> (INSERT INTO employees VALUES
    ("John" 1 1000 2003)
    ("Jack" 2 2000 2000)
    ("James" 2 1000 2010))
> (EXPORT employees foo.csv)
> (SELECT (emp_name dept_name)
   FROM (employees departments)
   WHERE ((yearly_salary salary) < 20000)
     AND (hired > 2005))
'(("James" "Marketing"))
> (SELECT (emp_name dept_name MIN(hired))
          FROM (employees departments))
'("Jack" "Marketing" 2000)
> (DELETE FROM employees WHERE emp_name = "James")
> (exit)
$ cat foo.csv
"John",1,1000,2003
"Jack",2,2000,2000
"James",2,1000,2010
```

3 Takuzu

`tic-tac-logic.hs`, **Haskell**

Takuzu (also known as Tic-Tac-Logic[3], Binairo, Binero, Tohu wa Vohu and Binary Puzzle) is a single player puzzle variation of the classic tic-tac-toe game. It is played on a rectangular grid with an even number of rows and columns.

The rules of the game are simple:

- Each cell of the grid should contain one of two marks, `X` or `O`.
- No more than two similar marks are allowed next to each other horizontally or vertically.
- Each row and column should have an equal number of `X`s and `O`s.
- Finally, no two rows or columns should be identical.

Write an *executable* Haskell program that reads Takuzu instances from the standard input and prints the solutions on the standard output. The given puzzle will always have a solution.

Requirements

- Your solution should be based on an iterative method.
- A list of tactics for solving an instance can be found here[4]. Your solution should include reasoning at least as powerful as the one described in Basic techniques 1-4.
- Your solution should not get stuck: if no progress can be made with tactics, it should begin trying each alternative for an empty cell.

Extra requirements

1. Provide testing functions for your code. (**2 points**)
2. From the same list of techniques[4], also include the following reasoning techniques:
   (a) Basic Technique 5. (**2 points**)
   (b) Advanced Technique 1. (**2 points**)
   (c) Advanced Technique 2. (**2 points**)
3. Use Haskell's unique features in clever ways. (**2 points**)
   (Make sure that you explain this part in your report.)

Input-Output

Each time your program will be solving a single instance of Tic-Tac-Logic. The first line of input contains the number of rows and columns of the instance. Each subsequent line describes the contents of each row, using `.` to denote an empty cell. You may assume that the grid's maximum size is 42×42, and your solution must solve it within 10 seconds.

Sample input, tic-tac.txt

```
6 6
X..O..
..OO.X
.OO..X
......
OO.X..
.X..OO
```

[3] http://www.conceptispuzzles.com/index.aspx?uri=puzzle/tic-tac-logic
[4] http://www.conceptispuzzles.com/index.aspx?uri=puzzle/tic-tac-logic/techniques

Use and output

```
$ ghc tic-tac-logic.hs
[1 of 1] Compiling Main      ( tic-tac-logic.hs, tic-tac-logic.o )
Linking tic-tac-logic ...
$ cat tic-tac.txt | ./tic-tac-logic
XOXOXO
OXOOXX
XOOXOX
OXXOXO
OOXXOX
XXOXOO
```

A larger example

```
16 22
...OO.X..X..O........X
........X.....X..X..O.
O..O.....O...O......O.
OO...XX..O.X..........
......X......O...O..XX
O......O....X.O...X...
.X.....OX...X...X.....
....O.X...X..O.O..O...
.O.................X..
X..XX.O.X....X........
X..O....X...O.........
.........O......X..O.O
..........X....XX.....
...........O........X.
.OO.........O..O...X..
OO...X..O..X.X..X..X..
```