Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών
`https://courses.softlab.ntua.gr/pl2/`

Γλώσσες Προγραμματισμού II

Αν δεν αναφέρεται διαφορετικά, οι ασκήσεις πρέπει να παραδίδονται στους διδάσκοντες σε ηλεκτρονική μορφή μέσω του συνεργατικού συστήματος ηλεκτρονικής μάθησης `moodle.softlab.ntua.gr`. Η προθεσμία παράδοσης θα τηρείται αυστηρά. Έχετε δικαίωμα να καθυστερήσετε το πολύ μία άσκηση.

Άσκηση 4 Erlang και property-based testing

Προθεσμία παράδοσης: 10/1/2021

Ο σκοπός της συγκεκριμένης άσκησης είναι να σας εξοικειώσει με τη γλώσσα προγραμματισμού Erlang και τη χρήση του εργαλείου PropEr για property-based testing. Ακολουθεί η εκφώνησή της στα αγγλικά.

A simple programming language is used for evaluating vector expressions. The syntax and the semantics of the language are defined below.

Vector language syntax

⟨*expr*⟩ ::= ⟨*vector*⟩
| {⟨*vector-op*⟩, ⟨*expr*⟩, ⟨*expr*⟩}
| {⟨*scalar-op*⟩, ⟨*int-expr*⟩, ⟨*expr*⟩}

⟨*vector*⟩ ::= [⟨*integer*⟩, . . .]

⟨*int-expr*⟩ ::= ⟨*integer*⟩
| {⟨*norm*⟩, ⟨*expr*⟩}

⟨*vector-op*⟩ ::= `add` | `sub` | `dot`

⟨*scalar-op*⟩ ::= `mul` | `div`

⟨*norm*⟩ ::= `norm_one` | `norm_inf`

Vector language semantics

- The binary vector operations are: addition, subtraction and a **non-aggregated** "dot product"-like operator which is just a pairwise multiplication of the vectors' elements.
- '`mul`' is multiplication and '`div`' is integer division of all vector elements with an integer.
- '`norm_one`' or "Taxicab norm" for vectors is defined as $\|\mathbf{x}\|_1 := \sum_{i=1}^{n} |x_i|$.
- '`norm_inf`' or "Maximum norm" for vectors is defined as $\|\mathbf{x}\|_\infty := \max_{i=1}^{n} |x_i|$.
- Integers are not bounded.

Evaluation rules

- The evaluation results in a vector, unless it fails.
- The evaluation **must fail** if:
  - An integer division by 0 is attempted.
  - Any vector in the input has a number of elements that is not between 1 and 100.
  - An expression is nesting deeper than 100 levels.
  - The sizes of vectors in a binary vector operation (i.e., '`add`', '`sub`', '`dot`') are not equal.

Evaluators

An evaluator for the vector language is an Erlang function that takes as parameter a vector expression ⟨*expr*⟩. The function should return the result of the evaluation: a vector, if the evaluation is successful, or the message "error", if the evaluation has failed.

If `evaluate/1` is such an evaluator function, examples of its use are given below:

```
1> evaluate([1,2,3,4,5]).
[1,2,3,4,5]
2> evaluate({'dot', [6,6,6], [7,7,7]}).
[42,42,42]
3> evaluate({'mul', {'norm_one', [1,-1,2,-2]}, [7,-7,7]}).
[42,-42,42]
4> evaluate({'div', 0, [1,2,3,4,5]}).
error
```

Task

The module `vectors.beam` (link to download) contains 50 implementations of evaluators for the vector language. Unfortunately, 46 of them have bugs...

Your task is to write *properties* that can be used to test the evaluators. Identify the evaluators that do not conform to the specification, by giving an input, the expected output, and the buggy evaluator's output.

Interface of `vectors.beam`

The contents of `vectors.erl` (link to download) are shown in Fig. 1. As you can see, the function `vector/1` can be used to get the evaluator corresponding to the provided `Id`. (Evaluator IDs are integers between 1 and 50, inclusive.) The evaluators can also be called directly using the `vector_N/1` functions.

Expected interface of `bughunt.erl`

You are required to submit a file `bughunt.erl`, defining an Erlang module `bughunt`. Among other functions, this module should export a function `test/1` that takes as input an evaluator ID and returns one of the following:

- if the input is the ID of a correct evaluator, the atom `'correct'` must be returned.
- if the input is the ID of a buggy evaluator, the returned value must be a tuple of the form `{Input, ExpectedOutput, ActualOutput, Comment}`, where:

```
-module(vectors).
-export([vector/1, vector_1/1, ... vector_50/1]).

-type vector()    :: [integer(),...].
-type expr()      :: vector()
                   | {vector_op(),     expr(), expr()}
                   | {scalar_op(), int_expr(), expr()}.
-type int_expr()  :: integer()
                   | {norm_op(), expr()}.
-type vector_op() :: 'add' | 'sub' | 'dot'.
-type scalar_op() :: 'mul' | 'div'.
-type norm_op()   :: 'norm_one' | 'norm_inf'.

-spec vector(integer()) -> fun((expr()) -> vector() | 'error').
vector(Id) when Id > 0, Id < 51 ->
  Name = list_to_atom(lists:flatten(io_lib:format("vector_~p",[Id]))),
  fun ?MODULE:Name/1.

-spec vector_1(expr()) -> vector() | 'error'.
vector_1(Expr) -> %% ???
...
-spec vector_50(expr()) -> vector() | 'error'.
vector_50(Expr) -> %% ???
```

Figure 1: Contents of `vectors.erl`, defining the interface of `vectors.beam`.

- `Input` is an Erlang term of type `expr()`
- `ExpectedOutput` is the expected output when evaluating the input
- `ActualOutput` is the output returned by the buggy evaluator or the atom `'crash'` (if the evaluator crashes) and
- `Comment` is a string that shortly describes a probable cause for the bug (you can leave it empty if you are not sure about the bug)

Sample

Assume that there are two more hypothetical evaluators, with IDs 51 and 52. Assume that the former is correct and the latter does not support addition. Your **`bughunt`**`:test/1` function should behave similarly to the following:

```
1> vectors:vector_51({'div', {'norm_inf', [-1, 5, 10]}, [1, 10, 100, 9999]}).
[0, 1, 10, 999]
2> bughunt:test(51).
correct
3> vectors:vector_52({'add', [1], [1]}).
error
4> vectors:vector_52({'sub', [1], [1]}).
[0]
5> bughunt:test(52).
{{'add', [1], [1]}, [2], error, "The operation 'add' is not supported."}
```