Heinrich-Heine-University Düsseldorf
Computer Science Department
Software Engineering and Programming Languages
Philipp Körner
Björn Ebbinghaus

Functional Programming – ST 2024
Reading Guide 01: Generative Testing

Timeline: It would be great if you could complete this unit by 22.04.2024.

1 Material

- REPL sessions, in order (recommended):
  - Material/repl2022-vertiefung/src/repl/21_transients.clj
  - Material/repl2022-vertiefung/src/repl/22_ebt.clj
  - Material/repl2022-vertiefung/src/repl/15_test_check.clj — Note: in this session, we show how a bug in Clojure's data structures was found. We use the old (bugged) version of Clojure and a compatible version of the test.check library for this. If you use leiningen, you might need to execute `lein downgrade 2.8.1` to get a compatible version. You can revert to a current version with `lein upgrade` afterwards (you should also edit the `project.clj` file to use a more recent Clojure version for the rest of the next reading guides). You might be able to set up a `deps.edn` file (see `project.clj` of the REPL session for the dependencies) and use the CLI tools instead.
- Book (optional): Getting Clojure, chapter 14
- Videos (optional, but very interesting):
  - Reid Draper: Powerful Testing with test.check `https://www.youtube.com/watch?v=JMhNINPo_g` (There is significant overlap with the REPL session. You should know about Transients before you watch this video.)
  - Gary Fredericks: Building test.check Generators `https://www.youtube.com/watch?v=F4VZPxLZUdA` (More details on how generators work internally and a slower-paced, more in-depth explanation of the more advanced ones).
  - John Hughes: Testing the Hard Stuff and Staying Sane `https://www.youtube.com/watch?v=zi0rHwfiX1Q` (Talk by a co-developer of the original library. Entertaining video. Uses the technique in Erlang to debug C code.)

2 Learning Outcomes

After completing this unit you should be able to

- use transients correctly.
- write example-based tests in Clojure.
- write generative tests in Clojure.
- discuss advantages and disadvantages of example-based and generative tests.

3 Highlights

- Transients: `transient`, `persistent!`, `conj!`, `disj!`, `assoc!`, `dissoc!`, `disj!`
- Testing: `deftest`, `is`, `are`
- test.check: generator-namespace, `for-all`, `quick-check`

4 Exercises

Exercise 1.1 (`test.check`)
In the files for the exercise you will find the namespace `unit09.edit-distance`. Have a look at the functions `levenshtein` and `levenschtein`.

a) Create a namespace in the test directory, that can access both functions.

b) Write a `test.check` property, which fails if two non-empty input-strings return different output for both functions. Which implementation is faulty? What is the error?

Note: You do not need to repair the faulty implementation.

c) Write a generator `user-generator`, which generates users at a university. A user is represented by a map, containing the first- and last name as well as the ID of the user. The ID consists of the first two characters of the first name, the first three characters of the last name and three digits. An example of a generated user is:

`{:first-name "John", :last-name "Wayne", :id "joway142"}`.

If a name is not long enough, as many characters as possible are used in the ID, for example one possible ID of „Rich X“ would be `rix666`

You can choose from a fixed set of names. For this purpose, you can use the name lists from `https://github.com/dominictarr/random-name`.

Exercise 1.2 (Shortest path problem)
Given a triangle in the form of a vector of vectors we want to find the shortest possible path from the top of the triangle to the bottom, such that the sum of the weights is minimized. In each step, only one adjacent field in the next lower row may be selected.

```
user=> (path [   [1]
                [2 4]
               [5 1 4]
              [2 3 4 5]])
7 ;; 1 + 2 + 1 + 3
user=> (path [    [3]
                 [2 4]
                [1 9 3]
               [9 9 2 4]
              [4 6 6 7 8]
             [5 7 3 5 1 4]])
20 ;; ; 3 + 4 + 3 + 2 + 7 + 1
```

Exercise 1.3 (Trampoline (4clojure No. 78) and recursion)
We have already seen that self-recursion in tail position using `recur` ensures that no additional stack frames are used per recursion step. This approach does not work if two (or more) functions call each other.

The higher-order function `trampoline` receives a function and an arbitrary amount of values. The function is then called with the given values as parameters. If the return value itself is a function again, it is called without parameters. As long as the resulting return values are functions, they will continue to be called, otherwise the value is returned by `trampoline`.

a) Implement a function that behaves like `trampoline`. In particular, the recursion should not use any more stack frames.

```
(letfn [(triple [x] (fn [] (sub-two (* 3 x))))
        (sub-two [x] (fn [] (stop? (- x 2))))
        (stop? [x] (if (> x 50) x (fn [] (triple x))))]
  (my-trampoline triple 2))
=> 82

(letfn [(my-even? [x] (if (zero? x) true (fn [] (my-odd? (dec x)))))
        (my-odd? [x] (if (zero? x) false (fn [] (my-even? (dec x)))))]
  (map (partial my-trampoline my-even?) (range 6)))
=> [true false true false true false]
```

Note: You can use `fn?` to check if a value is a function or not.

Additonal note: `partial` is a higher order function, which receives a function `f` and a part of its arguments $a_1, \ldots, a_i$ and returns a function which accepts further parameters $a_{i+1}, \ldots, a_n$ ($i \leq n$) and then calls `f` with parameters a_1 to a_n. `letfn` is a special `let` for functions.

b) What do you have to do if the result of the entire calculation happens to be a function itself?

Questions

If you have any questions, please contact Philipp Körner (`p.koerner@hhu.de`).