Computer Science 3MI3 – 2020 homework 9

Adding "guarded commands" to Clojure

Mark Armstrong

November 28th, 2020

Contents

Introduction

In his 1975 paper Guarded commands, nondeterminacy and formal derivation of programs, Edsger W. Dijkstra proposed a foundation for an imperative language different from the commonly used branching and iterating constructs; the *guarded commands*, along with control structures operating on them.

The guarded command language is especially interesting in comparison to the languages we have developed in that it is inherently *non-deterministic*.

In this homework, we familiarise ourselves with the guarded command control constructs by implementing them in Clojure, using macros.

Boilerplate

Submission procedures

Submission method

Homework should be submitted to your McMaster CAS Gitlab respository in the `cs3mi3-fall2020` project.

Ensure that you have **pushed** the commits to the remote repository in time for the deadline, and not just committed to your local copy.

Naming requirements

Place all files for the homework inside a folder titled `hn`, where `n` is the number of the homework. So, for homework 1, use the folder `h1`, for homework 2 the folder `h2`, etc. Ensure you do not capitalise the `h`.

Unless otherwise instructed in the homework questions, place all of your code for the homework in a single file in the `hn` folder named `hn.ext`, where `ext` is the appropriate extension for the language used according to this list:

- For Scala, `ext` is `sc`.
- For Prolog, `ext` is `pl`.
- For Ruby, `ext` is `rb`.
- For Clojure, `ext` is `clj`.

If multiple languages are used in the homework, submit a `hn.ext` file for each language.

If the language supports multiple different file extensions, you must still follow the extension conventions above.

Incorrect naming of files may result in up to a 10% deduction in your grade.

Do not submit testing or diagnostic code

Unless you are instructed to do so in the homework questions, **you should not submit testing code with your homework submission**.

This includes

- any `main` function,
- any `print` statements which output information **that is not directly requested as console output in the homework questions**.

If you do not wish to remove diagnostic print statements manually, you will have to find a way to ensure that they disabled in your final submission.

For instance, by using a wrapper on the print function or macros.

Due date and allowance for technical difficulties

Homework is due on the second Sunday following its release, by the end of the day (midnight). Submissions past 00:00 may not be considered.

If you experience technical difficulties leading up to the submission time, please contact Mark **ASAP** with the details of the problem and, if possible, attach the current state of your homework to the communication. This information will help ensure we are able to accept your submission once the technical difficulties are resolved.

Proper conduct for coursework

Individual work

Unless explicitely stated in the homework questions, all homework in this course is intended to be *individually completed*.

You are welcome to discuss the content of the homework in the public forum of the class Microsoft Teams team homework channel, though obviously solutions or partial solutions should not be posted or described.

Private discussions about the homework cannot reasonably be forbidden, but such discussions should follow the same guidelines as public discussions.

Inappopriate collaboration via private discussions which is later discovered by course staff may be considered academic dishonesty.

When in doubt, make the discussion private, or report its contents to the course staff by making a note of it in your homework.

To clarify what is considered appropriate discussions of homework content, here are some examples:

1. Discussing the language features introduced or needed for the homework.
    - Such as relevant builtin datatypes and datatype definition methods and their general use.
    - Code snippets that are not partial solutions to the homework are welcome and encouraged.
2. Questions of the form "What is meant by `x`?", "Does `x` really mean `y`?" or "Is there a mistake with `x`?"

- Of course, questions of those form which would be answered by partial solutions are not considered appropriate.

3. Questions or advice about errors that may be encountered.

    - Such as "If you see a `scala.MatchError` you should probably add a catch-all `_` case to your `match` expressions."

Language library resources

Unless explicitely stated in the questions, it is not expected that you will use any language library resources in the homeworks.

Possible exceptions to this rule include implementations of datatypes we discuss in this course, such as lists or options/maybes, if they are included in a standard library instead of being builtin.

Basic operations on such types would also be allowed.

- For instance, `head`, `tail`, `append`, etc. on lists would not require explicit permission to be used.
- More complex operations such as sorting procedures would require permission before you used them.

Additionally, the standard *higher-order* operations including `map`, `reduce`, `flatten`, and `filter` are permitted generally, unless the task is to implement such a higher-order operator.

Part 0.1: An introduction to guarded commands

A very brief and informative article discussing the guarded command by Jerrold L. Wagener is freely available from the ACM digital library.

A *guarded command* consists of a (presumably boolean valued) *guard* along with a *command* (an expression or statement of the language).

We write a set or sequence of guarded commands as

```
B   S
B   S

B   S
```

(where each `B` is the guard for the command `S`).

By itself, a set of guarded commands is not a control structure. Instead, we introduce special constructs which operate on sets of guarded commands to form a control structure.

The `if` construct, when applied to a set of guarded commands, as in

```
if
  B    S
  B    S

  B    S
```

selects any command whose guard evaluates to true, and executes that command. If no command is true, it does nothing.

The `do` construct, when applied to a set of guarded commands, as in

```
do
  B    S
  B    S

  B    S
```

also selects any command whose guard evaluates to true and executes that command, but it *continues to do so* until no guard is true.

We will be interested in versions of the `if` and `do` constructs which *nondeterministically* select which command to evaluate. This can be accomplished by making use of functions which act *randomly*.

The appeal of this nondeterminism is that it forces the programmer to be certain that their programs behaviour does not depend upon the ordering of the commands; instead, the guards must be made explicit enough to ensure that their command is only executed in the correct context.

Part 0.2: Representing guarded commands in Clojure

We can easily write guarded commands in Clojure as a record consisting of the `guard` and the `command`.

```
(defrecord GuardedCommand [guard command])
```

We can create an instance of this record as in

```
(GuardedCommand. '(> x 5) '(- x 1))
```

We can access the fields of these records as if they were maps, or using syntax based on Java field accessors. That is, given a `GuardedCommand` instance `grd-cmd`, we can write e.g., `(:guard grd-cmd)` or `(.command grd-cmd)`.

```
(let [grd-cmd (GuardedCommand. '(> x 5) '(- x 1))]
  (printf "The guard of this command is %s\n" (:guard
  ↪  grd-cmd))
  (printf "The command of this command is %s\n" (.command
  ↪  grd-cmd)))
```

So we will operate on lists or vectors of `GuardCommands`.

Part 0.3: An example construct for using guarded commands

To get you started, we define here a *deterministic* `if` construct operating on guarded commands.

This construct differs from the one you will be tasked to define in that it always chooses the first command in the sequence whose guard is true (instead of nondeterministically/randomly selecting a command.)

```
(defn first-allowed-command
  "Find the first command in a sequence of guarded `commands`
whose `.guard` evaluates to a truthy value and return its
↪  `.command`.
Returns `nil` if none of the guards are satisfied."
  [commands]
  ;; If the `commands` list is empty, "do nothing" by
  ↪  returning `nil`.
  (if (empty? commands) nil
      ;; Otherwise, deconstruct the `commands` list into
      ;; the first `command` and the `rest`.
      (let [[command & rest] commands]
        ;; Diagnostic print statement, if needed.
        ;(printf "Checking command %s with guard %s and
        ↪  command %s\n" command (.guard command) (.command
        ↪  command))
        ;; Now check the `guard`, and if it's satisfied,
        ↪  return the first `command`.
```

```
        (if (eval (.guard command)) (.command command)
            ;; Otherwise, continue to check the `rest` of the
            ↪  guarded commands.
            (first-allowed-command rest)))))

(defmacro guarded-deterministic-if
  "Given a sequence of `GuardedCommands`, `commands`,
select the first guarded command whose `.guard` evaluates
to a truthy value and evaluate its `.command`."
  [& commands]
  ;; The body must be quoted, so that nothing is evaluated
  ↪  until runtime.
  `(eval ;; Evaluate...
    (first-allowed-command ;; ...the command returned by
     ↪  first-allowed-command...
     [~@commands]))) ;; to which we pass a vector of the
      ↪  commands.
;; The ~@ applied to `commands` here "splices" the elements of
↪  `commands` into place here.
;; That is, each element of the sequence `commands` is
↪  inserted here in order.
;; But not literally as a sequence (between parentheses or
↪  brackets.) Hence we wrap in [].
;; The use of the [] is actually quite particular; using a
↪  quoted list, '(...), would not work.
;; Because the guarded commands within would be treated as
↪  sequences instead of records.
```

As an example, we use this form to define a `max` operation.

```
(defn max [x y]
  (guarded-deterministic-if
   ;; For variables to maintain their meaning within a quoted
   ↪  list,
   ;; use the special backtick ` quote and unquote the
   ↪  variables with ~.
   (GuardedCommand. `(>= ~x ~y) x)
   (GuardedCommand. `(>= ~y ~x) y)))
```

Part 1: Sequence of commands whose guards are satisfied [20 points]

Create a function `allowed-commands` which, given a sequence of `GuardedCommands` (the record type defined in Part 0.2) produces a list of commands whose guard is satisfied (evaluates to a truthy value.)

For instance,

```
(let [x 10
      y 10]
  (allowed-commands
   [(GuardedCommand. `(>= ~x ~y) `(printf "%s is greater than
   ↪  or equal to %s" ~x ~y))
    (GuardedCommand. `(=  ~x ~y) `(printf "%s is equal to %s"
    ↪  ~x ~y))
    (GuardedCommand. `(<= ~x ~y) `(printf "%s is less than or
    ↪  equal to %s" ~x ~y))]))
```

should return a sequence of all three of the commands (since `x` is equal to `y` here.) Whereas

```
(let [x  5
      y 10]
  (allowed-commands
   [(GuardedCommand. `(>= ~x ~y) `(printf "%s is greater than
   ↪  or equal to %s" ~x ~y))
    (GuardedCommand. `(=  ~x ~y) `(printf "%s is equal to %s"
    ↪  ~x ~y))
    (GuardedCommand. `(<= ~x ~y) `(printf "%s is less than or
    ↪  equal to %s" ~x ~y))]))
```

should return a sequence containing only the last command (since `x` is strictly less than `y` here.)

(Refer to the `first-allowed-command` function defined in part 0.3 as a possible starting point for your `allowed-commands` function. Other approaches are permitted and encouraged, though.)

Part 2: A nondeterministic `if` expression for guarded commands [15 points]

Define a macro for the nondeterministic `if` construct called `guarded-if`.

It should take a sequence of `GuardedCommand` instances, randomly pick one whose guard is true, and execute its command.

The `rand-nth` function documented here, which picks a random element out of a sequence, may be of use here (in conjunction with your function from part 1.)

Part 3: A nondeterministic `do` expression for guarded commands [15 points]

Now define a macro for the nondeterministic `do` construct called `guarded-do`. In contrast to the `guarded-if` macro, this construct should continue evaluating commands until none of the guards are true.

Part 4: GCD [10 points]

Use the guarded command constructs you have defined to define a function `gcd` to find the greatest common denominator of two integers.

The intention for this part is that you use the `if` construct from part 1 and recursion to define the GCD function. A version using iteration (the `do` construct) is given as a bonus.

Note that the iterative algorithm for the GCD using guarded commands is very well known; it was the first example used by Dijkstra in his presentation of the language. You can see this algorithm in Wagener's paper referenced above. It should be relatively simple to translate this to a recursive algorithm.

Part 5: GCD by iteration [5 bonus points]

Define the function `gcd-iter` which calculates the GCD of two integers using iteration (the `do` construct.)

The challenge to this part is not the algorithm; instead, it is the use of mutable variables, which we have not shown in Clojure.

Testing

:TODO: