Computer Science 3MI3 – 2020 homework 8

Visualising the call stack in Clojure

Mark Armstrong

November 19th, 2020

Contents

Introduction

In this homework, we provide you with a new Clojure special form which can be used to create functions which automatically output a visualisation of their calling context, and hence their stack usage, as they run. You are tasked with using this new special form to implement several simple recursive functions both in a "naive" way and a tail recursive way.

See Part 0.1 for an overview of "naive" and tail recursion.

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: "Naive" recursion vs. tail recursion

As discussed during the week 9 tutorials, in general, we know that recursion is less efficient than iteration (looping) as a control structure due to the requirement of a *stack frame* for each recursive call.

For instance, consider this "naively" defined factorial function.

```
(defn factorial [n]
  (cond
    (= n 0) 1
    (> n 0) (* n (factorial (- n 1)))
    :else (throw (Exception. "Trying to calculate factorial of
    ↪  a negative number."))))
```

If we "unwind the recursion" for a call to the above function, we see that at each recursive step, there is more and more "work to be done" added outside the recursive call. When we do reach a base case, we then have to return back to each call, doing the remaining work along the way.

```
(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 (* 1 (factorial 0))))
```

```
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

This work to be done has to be remembered somehow, and in principle this means that the stack frame for each call must be maintained until that call returns.

In contrast to the "naive" approach above, a *tail recursive* definition can be constructed so that *no work* remains to be done upon returning from the recursive call. Because of this fact, if the language implementation chooses to do so, the stack frames can be reused, and we can return to the initial caller when we reach the base case, instead of returning back through each recursive caller.

```
(defn factorial-tr [n]
  (defn fact-iter [n collect]
    (cond
      (= n 0) collect
      (> n 0) (fact-iter (- n 1) (* collect n))
      :else (throw (Exception. "Trying to calculate factorial
      ↪  of a negative number."))))
  (fact-iter n 1))
```

If we visualise a call to this function, we can observe that the calling context is constant (rather than growing in size), implying that the memory requirements on the stack are also constant.

```
(factorial-tr 3)
(fact-iter 3 1)
(fact-iter 2 3)
(fact-iter 1 6)
(fact-iter 0 6)
6
```

Tail recursion is then more efficient than general recursion, and is in fact equivalent to looping or iterating constructs. The ideas behind an iterative implementation and a tail recursive implementation are also usually quite similar.

In this homework, we provide you with a new Clojure special form which can be used to create functions which automatically output a visualisation of their calling context, and hence their stack usage, as they run. You are

tasked with using this new special form to implement several simple recursive functions both in a "naive" way and a tail recursive way.

Part 0.2: A new special form to automate visualisation of the stack

In this section, we define the new special form used in this assignment, called "`unwindrec`".

You may read about its definition if you are interested. If you prefer though, skip to the next section, Part 0.3: Using the new special form *which shows you how to use `unwindrec`.
It's not required that you understand all the code in this section.

We need a way to replace a value in a string representation of a Clojure term with a different value. Specifically, this will be used to replace variable names with their values, and to replace the placeholder `rec` with recursive calls or the results of recursive calls.

```
(defn replace-value-in-termstring
  "Replace all occurrences of `value` with `replacement`
within a `string` which is assumed to Clojure term.

Since the `string` is assumed to be a Clojure term,
this replaces the `value` only if it surrounded by spaces,
↪  parentheses,
braces or brackets (i.e., not if it is a substring of some
↪  subterm.)
One use of this is to replace a variable's name with its
↪  value,
as in `(replace-value-in-termstring 'var var string)`."
  [value replacement string]
  (clojure.string/replace
   string
   ;; The first group of this pattern matches the beginning of
   ↪  the string,
   ;; whitespace, or an opening parenthese/brace/bracket.
   ;; The last group matches the end of the string,
   ↪  whitespace, or
   ;; a closing parenthese/brace/bracket.
```

```
   (re-pattern (str "(\\A|\\s|\\[|\\{|\\()(" value
   ↪  ")(\\)|\\}|\\]|\\s|\\z)"))
   (str "$1" replacement "$3")))
```

We actually need a version of the above which replaces multiple values.

```
(defn replace-values-in-termstring
  "Replace all occurrences of the eleents of `values`
with the corresponding elements of `replacements`
within a `string` which is assumed to Clojure term.

This is the multiple replacement version of
↪  `replace-value-in-termstring`.
Note that values are replaced from in order,
and if an earlier replacement adds to the termstring
a value later in the list that value will also be replaced
(but not vice versa.)

Since the `string` is assumed to be a Clojure term,
this replaces the `value` only if it surrounded by spaces,
↪  parentheses,
braces or brackets (i.e., not if it is a substring of some
↪  subterm.)
One use of this is to replace a variable's name with its
↪  value,
as in `(replace-value-in-termstring 'var var string)`."
  [values replacements string]
  ;; `values` and `replacements` are assumed to be the same
  ↪  length,
  ;; but we check both are non-empty.
  (if (and (seq values) (seq replacements))
    ;; Deconstruct the lists.
    (let [[v & vs] values
          [r & rs] replacements]
      (replace-values-in-termstring vs rs
                                    (replace-value-in-
                                    ↪  termstring v r
                                    ↪  string)))
    ;; If `values` or `replacements` is empty, just return the
    ↪  string.
    string))
```

We'll need to use the above for each element of the `args` list, replacing each element with its current value, and also replacing instances of the function name (with `f` and the `context` argument.)

We'll also need to replace the recursive call(s) with a placeholder before we pass it along to the recursive call, so that the recursive call can tell where it was called from in the string.

This method to find the first closing parenthese that is not matched in a string will be necessary to tell where to replace up to. I don't see a way that regular expressions could be used for this, as there's no good way to specify "match up to a closing parenthese *that does not have an opening parenthese match*". This function instead uses a helper which keeps track of the number of unmatched opening parentheses as we walk through the string.

```
(defn closing-paren-in-string
  "Given a string `s`, return an two element list consisting
  ↪  of
the portion of `s` up to (and including) the first closing
↪  parenthese
that does not have a matching opening parenthese, and the
↪  remainder of `s`."
  [s]
  (letfn [(nth-closing-paren-in-string [s openings]
            (cond
              ;; If `s` has less than 2 characters, then even
              ↪  if it is a closing paren,
              ;; it's the first one and belongs in the first
              ↪  list.
              (> 2 (count s)) `(~(apply str s) "")
              :else
              ;; Otherwise, decompose `s` to check its first
              ↪  character.
              (let [[c & cs] s]
                (cond
                  ;; If it's an opening paren, increment
                  ↪  `openings` in the recursive call
                  ;; and prepend `c` to the returned first
                  ↪  string.
                  (= c \() (let [[before after]
                  ↪  (nth-closing-paren-in-string cs (+
                  ↪  openings 1))]
```

```
                    `(~(str c before) ~(apply str
                    ↪  after)))
                ;; If it's a closing paren, either split the
                ↪  string here if `openngs` is 0,
                ;; or decrement `openings` in the recursive
                ↪  call. And prepend `c` to the first
                ↪  string.
                (= c \)) (if (= openings 0)
                           `(~(str c) ~(apply str cs))
                           (let [[before after]
                           ↪  (nth-closing-paren-in-string
                           ↪  cs (- openings 1))]
                             `(~(str c before) ~after)))
                ;; Otherwise, just make the recursive call
                ↪  and prepend `c` to the returned first
                ↪  string.
                :else     (let [[before after]
                ↪  (nth-closing-paren-in-string cs
                ↪  openings)]
                            `(~(str c before) ~after))))))]
  ;; Start out with 0 opening parentheses seen.
  (nth-closing-paren-in-string s 0)))
```

Now, we can replace all instances of a call to a function by splitting the string at that function name, then further splitting the second part at the first unmatched closing parenthese, and replacing the chunk for the function call with a placeholder.

```
(defn replace-call-in-termstring
  "Replace all calls to a function whose name is given by `f`
in a string representing "
  [f replacement string]
   ;; Find the first occurrence of `f` preceded by an opening
   ↪  parenthese.
   (let [m (re-find (re-pattern (str "(.*)\\((" f "\\s.*)"))
   ↪  string)]
     ;; The match m will be nil if no match was found.
     ↪  Otherwise,
     ;; because the pattern has two groups, it will be a
     ↪  vector
     ;; of the whole match, the portion before the `f` call,
```

```
    ;; and the `f` call and remainder of the string.
    ;; Note the opening parenthese is not in either group.
    (if m
      ;; Get the parts of m.
      (let [[whole before call-and-after] m
            ;; Separate `call-and-after` at the first
            ↪  unmatched closing parenthese.
            [callbody after] (closing-paren-in-string
            ↪  call-and-after)]
        (str before replacement after))
      ;; If m is null, just return the string as is.
      string)))
```

Now we are finally ready to define our new special form. It constructs a recursive function of the same form we used for `factorial` and `factorial-tr`, carrying out the work of manipulating the "context string" automatically.

```
(defmacro unwindrec
  "Define a simple recursive function which prints out the
unwinding of the recursion as it runs.

The name of the function is given by `name`, and its arguments
↪  by `args`.

It is assumed that the function has a single base case,
guarded by condition `basecond` and with body `basebody`.
`basebody` should not contain any recursive calls;
the printing will not work properly otherwise.

It is further assumed that the function has a single recursive
↪  case,
guarded by condition `reccond` and with body `recbody`.
`recbody` should contain exactly one recursive call to `name`
somewhere in its body. The printing may not work properly
↪  otherwise.

The final, optional, argument `elsebody` is used as the body
if neither the `basecond` or `reccond` is satisfied.
"
  ([name args basecond basebody reccond recbody]
```

```
 ;; If `elsebody` is not provided, substitute `nil` instead.
 `(unwindrec ~name ~args ~basecond ~basebody ~reccond
 ↪  ~recbody nil))
([name args basecond basebody reccond recbody elsebody]
 ;; Define the function `name` taking arguments `args`.
 `(defn ~name ~args
    ;; This local function `f` actually implements `name`.
    ;; It has an additional argument, a string giving the
    ↪  context of
    ;; the call. This string should contain a substring
    ↪  "rec"
    ;; which is the point in the context at which the call
    ↪  was made.
    (letfn
        [(~'f [~'context ~@args]
          ;; Use the provided base case and recursive case
          ↪  conditions.
          (cond
            ;; In the base case, evaluate `basebody`.
            ↪  Substitute that value
            ;; for "rec" in the `context` and print the
            ↪  context,
            ;; then return the value.
            ~basecond (let [~'value ~basebody]
                        (println
                        ↪  (replace-value-in-termstring
                                '~'rec
                                ~'value
                                ~'context))
                        ~'value)
            ;; In the recursive case, replace "rec" in the
            ↪  `context`
            ;; with the recursive case body, and replace all
            ↪  argument names
            ;; with their current values.
            ~reccond  (let [~'this-context-with-call
            ↪  (replace-values-in-termstring
                                                    '~args
```

```
                                        (map
                                        ↪  str
                                        ↪  ~args)

                                        ↪  (replace-
                                        ↪  value-
                                        ↪  in-
                                        ↪  termstring

                                         ↪  '~'rec
                                        (str
                                         ↪  '~recbody)

                                         ↪  ~'context))
    ;; Also construct a new context
    ↪  `this-context-with-rec` to
    ↪  be
    ;; passed in the recursive call
    ↪  by replacing
    ;; the recursive call in
    ↪  `this-context-with-call`
    ↪  with "rec".
    ~'this-context-with-rec
    ↪  (replace-call-in-termstring
                                 '~name
                                "rec"
                                ~'this-
                                 ↪  context-
                                 ↪  with-
                                 ↪  call)]
(println ~'this-context-with-call)
;; Now, actually make the recursive
↪  call, but first
;; walk the `recbody` to replace
↪  `name` with `f` and
;; the additional argument
↪  `this-context-with-rec`
;; (remember that `f` is
↪  implementing `name`.)
```

```
                    (let [~'result
                    ↪  ~(clojure.walk/prewalk-replace
                    ↪  {name '(partial f
                    ↪  this-context-with-rec)} recbody)
                          ~'this-context-with-result
                           ↪  (replace-value-in-
                           ↪  termstring '~'rec (str
                           ↪  ~'result) ~'context)]
                      ;; Print out the context again
                      ↪  with the result in place,
                      ↪  unless
                      ;; the whole context is just the
                      ↪  result (this is so
                      ;; just the result does not get
                      ↪  printed over and over.)
                      (when (not=
                      ↪  ~'this-context-with-result
                      ↪  (str ~'result))
                        (println
                        ↪  ~'this-context-with-result))
                      ~'result))
            ;; If neither `basecond` nor `reccond` was
            ↪  satisfied,
            ;; evaluate `elsebody`. This does no printing.
            :else ~elsebody))]

      ;; Before calling `f`, print out the originating call.
      (println (replace-values-in-termstring
                '~args
                (map str ~args)
                (str "(" (clojure.string/join " " '(~name
                ↪  ~@args)) ")")))
      ;; Then call `f`.
      (~'f "rec" ~@args)))))
```

Part 0.3: Using the new special form

Here, we show how to use this new special form to define simple recursive functions such as factorial.

First, the "naive" recursive factorial.

```
(unwindrec factorial [n]
           (= n 0) 1
           (> n 0) (* n (factorial (- n 1)))
           (throw (Exception. "Trying to calculate factorial
           ↪  of a negative number.")))
```

And then the tail recursive variant. Note that we must again wrap the helper function which takes two arguments in another `defn` that defines the function taking one argument. (In this case, the `factorial-tr-helper` function will be defined globally, because `unwindrec` expands to a `defn` instead of a `letfn`, but that's okay.)

```
(defn factorial-tr [n]
  (unwindrec factorial-tr-helper [n collect]
             (= n 0) collect
             (> n 0) (factorial-tr-helper (- n 1) (* collect
             ↪  n))
             (throw (Exception. "Trying to calculate factorial
             ↪  of a negative number.")))
  (factorial-tr-helper n 1))
```

Part 1: Exponent [10 marks]

Similar to the `factorial` and `factorial-tr` functions defined using the `unwindrec` form in Part 0.3, define two functions `exponent` and `exponent-tr` which implement the exponent/power function on natural numbers (meaning that negative exponents are not supported.)

The full definitions of `unwindrec` and the helper functions it uses can be found here. You may either copy the contents of this file into yours, or import it, for instance by placing it in the same directory as your `h8.clj` and writing `(load-file "./unwindrec.clj")` at the top of your file.

The `exponent` function should use "naive" recursion on the second argument, whereas `exponent-tr` should use tail recursion (still on the second argument.)

For instance, `(exponent 2 3)` should return `8`, since $2^3 = 8$.

Part 2: Sum of a list [10 marks]

Now, use the `unwindrec` form to define functions `sumlist` and `sumlist-tr` which sum up the elements of a list.

For instance, `(sumlist '(0 1 2 3))` should return `6`.

Part 3: Flatten a list [20 marks]

Use `unwindrec` again more to define functions `flattenlist` and `flattenlist-tr` which flatten a list of lists to a list, by concatenating the lists.

For instance, `(flattenlist '((1 2) (3 4) (5 6 7 8)))` should return `(1 2 3 4 5 6 7 8)`.

Part 4: Prefixes of a list [20 marks]

Use `unwindrec` once more to define functions `postfixes` and `postfixes-tr` which, given a list `l`, return all the sublists which are postfixes of `l`, including `l` itself and the empty list, in decreasing order of length.

For instance, `(postfixes '(1 2 3 4 5 6))` should return `((1 2 3 4 5 6) (2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) ())`.

Make sure that the empty list is considered a postfix.

Note that you may need to write the list containing the empty list as part of your code; this can be written `'(())`.

You may also need to write the list containing the value of a variable, say for instance a variable `x`. You will find that if you write `'(x)`, `x` is not evaluated and so is not replaced by its value. Instead, you must write something like `` `(~x) ``. The backtick `` ` `` is a special kind of quote that allows for parts of the quoted expression, marked with tildes `~`, to be evaluated.

Testing

:TODO: