Computer Science 3MI3 – 2020 assignment 3

A representation of Dijkstra's guarded command language

Mark Armstrong

December 2, 2020

Contents

Introduction

This assignment

Updates and file history

December 1st

- Initial version posted.

Boilerplate

Documentation

In addition to the code for the assignments, you are required to submit (relatively light) documentation, along the lines of that found in the literate programs from lectures and tutorials.

- Those occasionally include a lot of writing when introducing concepts; you do not have to introduce concepts, so your documentation should be similar to the *end* of those documents, where only the purpose and implementation details of types, functions, etc., are discussed.

This documentation is not assigned its own marks; rather, 20% of the marks of each part of the assignment will be for the documentation.

This documentation **must be** in the literate style, with (nicely typeset) English paragraphs alongside code snippets; comments in your source code do not count. The basic requirement is

- the English paragraphs must use non-fixed width font, whereas
- the code snippets must use fixed width font.
- For example, see these lecture notes on Prolog:
  - `https://courses.cs.washington.edu/courses/cse341/98sp/logic/prolog.html`

But you are encouraged to strive for nicer than just "the basic requirement". (the ability to write decent looking documentation is an asset!

You are free to present your documentation in any of these formats:

- an HTML file,
  - (named `README.html`)
- a PDF (for instance, by writing it in LaTeX using the `listings` or `minted` package for your code blocks),
  - (named `README.pdf`), or
- rendering on GitLab (for instance, by writing it in markdown or Org)
  - (named `README.md` or `README.org`.)

If you wish to use another format, contact Mark to discuss it.

Not all of your code needs to be shown; only portions which are of interest are needed. Feel free to omit some "repetitive" portions. (For instance, if there are several cases in a definition which look almost identical, only one or two need to be shown.)

Submission procedures

The same guidelines as for homework (which can be seen in any of the homework files) apply to assignments, except for the differences below.

Assignment naming requirements

Place all files for the assignment inside a folder titled `an`, where `n` is the number of the assignment. So, for assignment 1, use the folder `a1`, for assignment 2 the folder `a2`, etc. Ensure you do not capitalise the `a`.

Each part of the assignments will direct you on where to save your code for that part. Follow those instructions!

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

Incorrect naming of files may result in up to a 5% deduction in your grade.

This is slightly decreased from the 10% for homeworks.

Proper conduct for coursework

Refer to the homework code of conduct available in any of the homework files. The same guidelines apply to assignments.

Part 0 – The guarded command language, *GCL*

This assignment involves representing a simple kind of *guarded command language*, which we call *GCL*, and a small extension to it which we call *GCLe*, which adds a notion of scope.

GCL

The syntax of *GCL* is given as

```
⟨expr⟩ ::= constant_integer | variable
         | ⟨expr⟩ ('+' | '*' | '-' | '÷') ⟨expr⟩

⟨test⟩ ::= ⟨expr⟩ ('=' | '<' | '>') ⟨expr⟩
         | ⟨test⟩ ('and' | 'or') ⟨test⟩


⟨stmt⟩ ::= 'skip'
         | variable '≔' ⟨expr⟩
```

```
        | ⟨stmt⟩ ';' ⟨stmt⟩
        | 'if' ⟨gc_list⟩
        | 'do' ⟨gc_list⟩

⟨gc⟩ ::= ⟨test⟩ ' ⇒ ' ⟨stmt⟩

⟨gc_list⟩ ::= { ⟨gc⟩ }
```

That is, the language consists of

- (integer) expressions built from integer constants, variable names, and the binary operations addition, multiplication, subtraction and division.
- (boolean) tests built from equality and inequality checks on expressions, along with `and` and `or`.
- statements, which may be
  - `skip`, the empty statement that does nothing,
  - assignment of an expression to a variable,
  - the composition of two statements,
  - the "choice" construct `if` applied to a list of guarded commands,
  - the "iteration" construct `do` applied to a list of guarded commands,
- and guarded command lists, which are a sequence of zero or more guarded commands,
  - where a guarded command consists of a (boolean) test and a statement.

For this language, we use the same notion of (memory) state as in the beginning of the notes on the *WHILE* language: a map or function from variable names to integers. **We assume for this language that variables are always initialised to 0**.

The semantics of the expressions, tests and the `skip`, assignment and composition statements are intended to be similar to those of *WHILE* as described in lecture.

The semantics of the `if` and `do` constructs on guarded command lists are as noted in homework 9, which discussed the guarded command. One important note: in both cases, if the guarded command list is empty, the result should be to "do nothing".

GCLe

The language *GCLe* is obtained from *GCL* by adding these productions to grammer.

$\langle$`program`$\rangle$ ::= $\langle$`globals`$\rangle$ $\langle$`stmt`$\rangle$

$\langle$`globals`$\rangle$ ::= `'global' { variable }`

$\langle$`stmt`$\rangle$ ::= `'local' variable 'in'` $\langle$`stmt`$\rangle$

The intent is that a *program* now consists of a list of global variables followed by a statement, which we may call the "body" of the program.

Additionally, we add a new kind of statement for declaring local variables.

With these constructs in place, we may now discuss whether a given program is *well-scoped*; that is, if every variable used in the program is either

- a global variable, or
- a local variable declared by some wrapping `local` statement.

We will assume in the semantics that all programs are well-scoped, and we can make use of a more precise notion of memory state; a memory state is some mapping from *variables which are in scope* to values. Variables which are not in scope are not handled by such a memory state.

Part 1 – Representations of *GCL* and a small extension

In Ruby and in Clojure, create a representation of the language *GCL* described in part 0.

In Ruby, define the types `GCExpr`, `GCTest` and `GCStmt`, with the following subclasses.

- `GCExpr` has subclasses
  - `GCConst`, the constructor of which takes a single integer argument,
  - `GCVar`, the constructor of which takes a symbol for the variable name,

  - `GCOp`, the constructor of which has as its first two arguments are `GCExpr`'s and as its third argument a symbol, which is intended to be one of `:plus`, `:times`, `:minus` or `:div`.

- `GCTest` has subclasses

  - `GCComp`, the constructor of which has as its first two arguments `GCExpr`'s and as its third argument a symbol, which is intended to be one of `:eq`, `:less` or `:greater`,
  - `GCAnd` and `GCOr`, the constructors of which take as arguments two `GCExpr`'s.

- `GCStmt` has subclasses

  - `GCSkip`, the constructor of which (if it exists) takes no arguments.
  - `GCAssign`, the constructor of which takes as arguments a symbol for the variable name and a `GCExpr`.
  - `GCCompose`, the constructor of which takes two `GCStmt`'s as arguments,
  - `GCIf` and `GCDo`, the constructors of which take a list of `GCTest` and `GCStmt` pairs (pairs being lists of two elements.)

Wrap all of these definitions inside a `module` named `GCL`. (This is to avoid name clashes with definitions requested below.)

In Clojure, define *records* (documentation and examples here) for each kind of expression, test and statement (using the same naming as in Ruby.) There is no need to define the `GCExpr`, `GCTest` and `GCStmt` types themselves; only the subtypes as records.

Then, in Ruby, create a separate representation of the language *GCLe* described in part 0. Create a class `GCProgram` to represent programs, the constructor of which takes as its first argument a list of symbols for the global variable names, and as its second argument a `GCStmt`. Also add an additional subclass to `GCStmt`, `GCLocal`, the constructor of which takes as its first argument a symbol for the variable name and as its second argument a `GCStmt`. Wrap all of these definitions inside a `module` named `GCLe`.

Part 2 – A stack machine for *GCL* in Ruby

Within the `GCL` module, define a method `stack_eval` on `GCL`'s, which carries out the evaluation of a `GCStmt` using a stack machine.

The stack machine in question should really be a method taking three arguments:

1. the command stack (implemented using a list),
2. the results stack (implemented a list), and
3. the memory state (implemented using a lambda; that is, a block.)

The method should return an updated state (that is, another lambda/block.)

Part 3 – The small-step semantics of *GCL* in Clojure

Define in Clojure a function `reduce` which takes a *GCL* statement and a memory state (a function mapping symbols, representing the variable names, to integers) and performs *one step* of the computation, returning the remaining code to be run and the updated memory state.

Part 4 – The big-step semantics of *GCLe* in Ruby

This portion of the assignment should be done in the `GCLe` module created in part 1.

Begin by defining a method `wellScoped` which checks that all variables appearing within the body of a `GCProgram` (either in an expression or on the left side of an assignment) are *within scope* at the point of their use; that is, either the variable is one declared to be `global`, or there is a `local` statement for that variable wrapping the use.

- This method should take a `GCStmt` as its only argument, and return a boolean.
- Hint: This operation is similar to typechecking. Use your experience working with `typeOf` as a starting point.
  - Helper methods are always permitted.

Then define the semantics of the language, this time defining a method `eval` directly (without making use of a stack machine.) That is, define the *big-step* semantics of the language (remember that big-step semantics are called evaluation semantics.)

- This method also should take a `GCStmt` as its only argument. It should return a `Hash` mapping the `global` variable names to integers.

You may decide what the behaviour is for programs which do not initialise variables before their first use.

- Your choice may be judged in the marking.
  - It is suggested that such programs "fail gracefully", reporting an error that a variable was used before initialisation.
  - Otherwise, it's suggested that they behave as predictably as possible.

Part 5 – *GCLe* in Clojure

As a bonus, repeat part 4 in Clojure.

Place the code for this portion in a file `a3b.clj`.

This time, you may choose the underlying approach to the operational semantics (you do not have to use big-step semantics.)

Document this portion especially well, and include your own tests in a file `a3bt.clj`. This file should output the results of the tests when executed using `cat a3bt.clj | lein` from the command line.

Submission checklist

For your convenience, this checklist is provided to track the files you need to submit. Use it if you wish.

```
- [ ] Documentation; one of
  - [ ] README.html
  - [ ] README.pdf
  - [ ] README.md
  - [ ] README.org
- [ ] Code files
  - [ ] a3.rb
  - [ ] a3.clj
- [ ] Part 2 tests
  - [ ] a3p2_test.rb tests have passed! (No submission
  ↪  needed.)
- [ ] Part 3 tests
```

```
  - [ ] a3p3_test.clj tests have passed! (No submission
   ↪  needed.)
- [ ] Part 4 tests
  - [ ] a3p4_test.clj tests have passed! (No submission
   ↪  needed.)
- [ ] Part 5 (Bonus)
  - [ ] a3b.clj
  - [ ] a3bt.clj
```

Testing

:TODO: