Representation of Guarded Command Language

Khizar Siddiqui

December 22, 2020

Contents

1 Introduction

This documentation was created for Assignment 3 of 3MI3, it's purpose is to help understand the fundamentals behind how (and why) my program runs. Guarded Command Language is a language defined for predicate transformer semantics. We will attempt to implement this language using Ruby and Clojure.

2 Representation

Our representation builds on top of standard expressions like integers, variables, and operations of expressions. We also introduce testing expressions like True/False, and boolean operations on them. Lastly we add our statements, which include a skip, variable assignment, composition, an if loop and a do loop (both loops taking a list of test and statement tuples).

2.1 Ruby

For Ruby, we will classify our 3 main terms of expressions, tests and statements as classes:

```
class GCExpr; end
class GCTest; end
class GCStmt; end
```

Then, we create classes for each subterm as subclasses that inherit their respective classes, note our skip, true and false classes take no arguments as they are base classes.

```
class GCConst < GCExpr
        attr_reader :val

        def initialize(x)
            unless x.is_a?(Integer)
                throw "Constructing a constant out of a non integer value"
            end
            @val = x
        end
    end

    class GCVar < GCExpr
        attr_reader :val

        def initialize(x)
            unless x.is_a?(Symbol)
                throw "Constructing a variable out of a non symbol value"
            end
            @val = x
        end
    end

    class GCOp < GCExpr
        attr_reader :val1
        attr_reader :val2
        attr_reader :op

        def initialize(x, y, z)
            unless x.is_a?(GCExpr) and y.is_a?(GCExpr) and [:plus, :minus, :times, :div].include? z
                throw "Constructing an operation out of incorrect values"
            end
            @val1 = x
            @val2 = y
            @op = z
        end
    end
```

```ruby
class GCComp < GCTest
    attr_reader :val1
    attr_reader :val2
    attr_reader :op

    def initialize(x, y, z)
        unless x.is_a?(GCExpr) and y.is_a?(GCExpr) and [:eq, :less, :greater].include?(z)
            throw "Constructing a comparison out of incorrect values"
        end
        @val1 = x
        @val2 = y
        @op = z
    end
end

class GCAnd < GCTest
    attr_reader :val1
    attr_reader :val2

    def initialize(x, y)
        unless x.is_a?(GCTest) and y.is_a?(GCTest)
            throw "Constructing AND with non GCExpr values"
        end
        @val1 = x
        @val2 = y
    end
end

class GCOr < GCTest
    attr_reader :val1
    attr_reader :val2

    def initialize(x, y)
        unless x.is_a?(GCTest) and y.is_a?(GCTest)
            throw "Constructing OR with non GCExpr values"
        end
        @val1 = x
        @val2 = y
    end
end

class GCTrue < GCTest
end

class GCFalse < GCTest
end

class GCSkip < GCStmt; end

class GCAssign < GCStmt
    attr_reader :var
    attr_reader :expr

    def initialize(x, y)
```

```ruby
        unless x.is_a?(Symbol) and y.is_a?(GCExpr)
            throw "Constructing an assignment out of incorrect values"
        end
        @var  = x
        @expr = y
    end
end

class GCCompose < GCStmt
    attr_reader :val1
    attr_reader :val2

    def initialize(x, y)
        unless x.is_a?(GCStmt) and y.is_a?(GCStmt)
            throw "Constructing a compose out of non GCStmt values"
        end
        @val1 = x
        @val2 = y
    end
end

class GCIf < GCStmt
    attr_reader :guards

    def initialize(x)
        unless checker?(x)
            throw "Constructing IF out of incorrect values"
        end
        @guards = x
    end

    def checker?(val)
        val.each do |v|
            unless v[0].is_a?(GCTest) and v[1].is_a?(GCStmt)
                return false
            end
        end
        return true
    end
end

class GCDo < GCStmt
    attr_reader :guards

    def initialize(x)
        unless checker?(x)
            throw "Constructing DO out of incorrect values"
        end
        @guards = x
    end

    def checker?(val)
        val.each do |v|
            unless v[0].is_a?(GCTest) and v[1].is_a?(GCStmt)
```

```
                    return false
                end
            end
            return true
        end
    end
```

2.2 Clojure

For Clojure, we define our subterms as records. We do not need to account for the main classes as we assume all inputs are well typed.

```
(defrecord GCConst [c])
(defrecord GCVar [v])
(defrecord GCOp [e1 e2 op])

(defrecord GCComp [e1 e2 op])
(defrecord GCAnd [t1 t2])
(defrecord GCOr [t1 t2])
(defrecord GCTrue [])
(defrecord GCFalse [])

(defrecord GCSkip [])
(defrecord GCAssign [v e])
(defrecord GCCompose [s1 s2])
(defrecord GCIf [guards])
(defrecord GCDo [guards])
```

3 Stack Machine

In Ruby, we define a stack machine implementation to carry out evaluation of statements. This stack machine comprises of a command stack where we store instructions to perform, a result stack to store temporary results and the memory which maps variables to integers.

```ruby
def stackEval(commands, result, memory)

        if commands.length() == 0
            return memory
        end

        command = commands.shift
        case command
        when GCConst
            result.unshift(command.val)
            stackEval(commands, result, memory)
        when GCVar
            stackEval(commands, result.unshift(memory.call(command.val)), memory)
        when GCOp
            commands.unshift(command.op).unshift(command.val1).unshift(command.val2)
            stackEval(commands, result, memory)
        when GCComp
            commands.unshift(command.op).unshift(command.val1).unshift(command.val2)
            stackEval(commands, result, memory)
        when GCAnd
            commands.unshift(:and).unshift(command.val1).unshift(command.val2)
            stackEval(commands, result, memory)
        when GCOr
            commands.unshift(:or).unshift(command.val1).unshift(command.val2)
            stackEval(commands, result, memory)
        when GCTrue
            result.unshift(true)
            stackEval(commands, result, memory)
        when GCFalse
            result.unshift(false)
            stackEval(commands, result, memory)
        when GCSkip
            stackEval(commands, result, memory)
        when GCAssign
            commands.unshift(:assign).unshift(command.expr)
            result.unshift(command.var)
            stackEval(commands, result, memory)
        when GCCompose
            commands.unshift(command.val2).unshift(command.val1)
            stackEval(commands, result, memory)
        when GCIf
            allguards = command.guards
            result.unshift(:ifstop)
            commands.unshift(:if)
            allguards.each do |guard|
                commands.unshift(guard[1])
                commands.unshift(:ifguard)
                commands.unshift(guard[0])
            end
```

```
    stackEval(commands, result, memory)
when GCDo
    allguards = command.guards
    result.unshift(:dostop)
    commands.unshift(command)
    commands.unshift(:do)
    allguards.each do |guard|
        commands.unshift(guard[1])
        commands.unshift(:doguard)
        commands.unshift(guard[0])
    end
    stackEval(commands, result, memory)
when :doguard
    bval = result.shift
    stmt = commands.shift
    if bval == true
        result.unshift(stmt)
    end
    stackEval(commands, result, memory)
when :do
    trueguards = []
    result.each do |guard|
        trueguards.push(guard)
        break if guard == :dostop
    end
    removestop = trueguards.pop
    if trueguards.length() > 0
        r = rand(0..trueguards.length()-1)
        commands.unshift(trueguards[r])
    else
        removedo = commands.shift
    end
    stackEval(commands, result, memory)
when :ifguard
    bval = result.shift
    stmt = commands.shift
    if bval == true
        result.unshift(stmt)
    end
    stackEval(commands, result, memory)
when :if
    trueguards = []
    result.each do |guard|
        trueguards.push(guard)
        break if guard == :ifstop
    end
    removestop = trueguards.pop
    if trueguards.length() > 0
        r = rand(0..trueguards.length()-1)
        commands.unshift(trueguards[r])
        stackEval(commands, result, memory)
    else
        stackEval(commands, result, memory)
    end
```

```
    when :plus
        val1 = result.shift
        val2 = result.shift
        result.unshift(val1+val2)
        stackEval(commands, result, memory)
    when :times
        val1 = result.shift
        val2 = result.shift
        result.unshift(val1*val2)
        stackEval(commands, result, memory)
    when :minus
        val1 = result.shift
        val2 = result.shift
        result.unshift(val1-val2)
        stackEval(commands, result, memory)
    when :div
        val1 = result.shift
        val2 = result.shift
        result.unshift(val1/val2)
        stackEval(commands, result, memory)
    when :eq
        val1 = result.shift
        val2 = result.shift
        result.unshift(val1==val2)
        stackEval(commands, result, memory)
    when :less
        val1 = result.shift
        val2 = result.shift
        result.unshift(val1<val2)
        stackEval(commands, result, memory)
    when :greater
        val1 = result.shift
        val2 = result.shift
        result.unshift(val1>val2)
        stackEval(commands, result, memory)
    when :and
        val1 = result.shift
        val2 = result.shift
        result.unshift(val1&&val2)
        stackEval(commands, result, memory)
    when :or
        val1 = result.shift
        val2 = result.shift
        result.unshift(val1||val2)
        stackEval(commands, result, memory)
    when :assign
        solution = result.shift
        variable = result.shift
        stackEval(commands, result, updateState(memory, variable, solution))
    end
end

def emptyState
    lambda { |c|
```

```
        0
    }
end

def updateState(sigma, x, n)
    lambda { |c|
        if (c == x)
            n
        else
            sigma.call(c)
        end
    }
end
```

The explanation is very straightforward for most of the terms. Constants/True/False move directly to the result stack, variables are searched inside the memory for a mapping. Operators (integer and boolean) are a little different where they evaluate each individual argument and using an operational symbol, pop the results from the result stack to evaluate further and push back into the result stack. Assignments simply add onto the memory with the specified variable and expression. If and Do, however, can get a little complicated. I implemented If by first pushing into the result stack an if symbol and the test statement of each guard, if it evaluates to True we then pop that from the result stack and push the statement of this true guard. Once the list is traversed, we pop out all the true statements at which point we randomly select one and add it to the command stack to be evaluated. Do is similar to If, except it has the entire Do loop on the stack right after the loops returns a True (to be performed again) If none are true, the Do loop is removed from the stack.

4 Small step semantics

We will use Clojure to now reduce our terms by one step (small step) and return the statement as well as its resulting state memory.

```
(defn drop-nth [n coll]
  (concat
    (take n coll)
    (drop (inc n) coll)))

(defn emptyState []
    (fn [x] 0))

(defn updateState [sig x n]
    (fn [c] (if (= c x) n ((sig) c))))

(defn reduce [some]
    (let [command (.stmt some)]
        (cond
        (instance? GCVar command)
            (Config. ((.sig some) (.v command)) (.sig some))
        (instance? GCOp command)
            (if (instance? GCConst (.e1 command))
                (if (instance? GCConst (.e2 command))
                    (cond
                    (= :plus  (.op command)) (Config. (+
                        (.c (.e1 command)) (.c (.e2 command))) (.sig some))
                    (= :minus (.op command)) (Config. (-
                        (.c (.e1 command)) (.c (.e2 command))) (.sig some))
                    (= :times (.op command)) (Config. (*
                        (.c (.e1 command)) (.c (.e2 command))) (.sig some))
                    (= :div   (.op command)) (Config. (/
                        (.c (.e1 command)) (.c (.e2 command))) (.sig some))
                    )
                        (let [newcommand (reduce (Config. (.e2 command) (.sig some)))]
                        (Config. (GCOp. (.e1 command) (.stmt newcommand) (.op command))
                            (.sig newcommand))
                        ))
                            (let [newercommand (reduce (Config. (.e1 command) (.sig some)))]
                            (Config. (GCOp. (.stmt newercommand) (.e2 command) (.op command))
                                (.sig newercommand))
                            ))
        (instance? GCComp command)
            (if (instance? GCConst (.e1 command))
                (if (instance? GCConst (.e2 command))
                    (cond
                    (= :eq      (.op command)) (Config. (=
                        (.c (.e1 command)) (.c (.e2 command))) (.sig some))
                    (= :less    (.op command)) (Config. (<
                        (.c (.e1 command)) (.c (.e2 command))) (.sig some))
                    (= :greater (.op command)) (Config. (>
                        (.c (.e1 command)) (.c (.e2 command))) (.sig some))
                    )
                        (let [newcommand (reduce (Config. (.e2 command) (.sig some)))]
                        (Config. (GCOp. (.e1 command) (.stmt newcommand) (.op command))
```

```
                    (.sig newcommand))
                ))
                    (let [newercommand (reduce (Config. (.e1 command) (.sig some)))]
                    (Config. (GCOp. (.stmt newercommand) (.e2 command) (.op command))
                        (.sig newercommand))
                    ))
(instance? GCAnd command)
    (if (or (instance? GCTrue (.t1 command)) (instance? GCFalse (.t1 command)))
        (if (or (instance? GCTrue (.t2 command)) (instance? GCFalse (.t2 command)))
            (if (and (instance? GCTrue (.t1 command)) (instance? GCTrue (.t2 command)))
                (Config. (GCTrue.) (.sig some))
                    (Config. (GCFalse.) (.sig some)))
                        (let [newcommand (reduce (Config. (.t2 command) (.sig some)))]
                        (Config. (GCAnd. (.t1 command) (.stmt newcommand)) (.sig newcommand))
                        ))
                            (let [newcommand (reduce (Config. (.t1 command) (.sig some)))]
                            (Config. (GCAnd. (.stmt newcommand) (.t2 command))
                                (.sig newcommand))
                            ))
(instance? GCOr command)
    (if (or (instance? GCTrue (.t1 command)) (instance? GCFalse (.t1 command)))
        (if (or (instance? GCTrue (.t2 command)) (instance? GCFalse (.t2 command)))
            (if (or (instance? GCTrue (.t1 command)) (instance? GCTrue (.t2 command)))
                (Config. (GCTrue.) (.sig some))
                    (Config. (GCFalse.) (.sig some)))
                        (let [newcommand (reduce (Config. (.t2 command) (.sig some)))]
                        (Config. (GCAnd. (.t1 command) (.stmt newcommand)) (.sig newcommand))
                        ))
                            (let [newcommand (reduce (Config. (.t1 command) (.sig some)))]
                            (Config. (GCAnd. (.stmt newcommand) (.t2 command))
                                (.sig newcommand))
                            ))
(instance? GCAssign command)
    (if (instance? GCConst (.e command))
        (Config. (GCSkip.) (updateState (.sig some) (.v command) (.c (.e command))))
            (let [newcommand (reduce (Config. (.e command) (.sig some)))]
            (Config. (GCAssign. (.v command) (.stmt newcommand)) (.sig newcommand))
            ))
(instance? GCCompose command)
    (if (instance? GCSkip (.s1 command))
        (Config. (.s2 command) (.sig some))
            (let [newcommand (reduce (Config. (.s1 command) (.sig some)))]
            (Config. (GCCompose. (.stmt newcommand) (.s2 command)) (.sig newcommand))
            ))
(instance? GCIf command)
    (if (= 0 (count (.guards command)))
        (Config. (GCSkip.) (.sig some))
            (let [random (rand-int (count (.guards command)))]
            (if (instance? GCTrue (get (get (.guards command) random) 0))
                (Config. (get (get (.guards command) random) 1) (.sig some))
                    (if (instance? GCFalse (get (get (.guards command) random) 0))
                        (Config. (GCIf. (drop-nth random (.guards command))) (.sig some))
                            (let [newcommand (reduce (Config. (get (get (.guards command)
                                random) 0) (.sig some)))]
```

```
                                    (Config. (assoc-in (.guards command) [random 0]
                                        (.stmt newcommand)) (.sig newcommand))
                                    )))
                    ))
        (instance? GCDo command)
            (if (= 0 (count (.guards command)))
                (Config. (GCSkip.) (.sig some))
                    (let [allguards (.guards command)
                        newguards (.guards command)
                        length    (count allguards)
                        ifguards  (loop [x 0]
                                        (when (< x length)
                                            (assoc-in (newguards) [x 1] (GCCompose.
                                                (get (get allguards x) 1) command))
                                            (recur (+ x 1))
                                            ))]
                    (Config. (GCIf. ifguards) (.sig some))
                    ))
        )))
```

Most of our code simply checks for the existence of a constant integer or a skip. If it exists for both arguments, the whole expression is evaluated, otherwise the non constant is reduced. If and Do is a little tricky however, If chooses a random guard from the list of guards and checks if it's test is True/False. If true, reduces the statement and if false, removes the guard from the list of guards and repeats. If it is neither true or false, it reduces the test and redoes the If with the new reduced test. If we traverse the entire list without encountering any true guard, we simply reduce down to a skip. Do is similar (though it has been a trouble to implement in Clojure) and we can utlize the If command as well. We simply take the list of guards and create a new If loop of those guards. However, for each guard statement we compose the entire while loop at the end of it. This way it keeps running the true guard statement until no true guards are left at which point it reduces to skip.

5 Big step semantics

We start off by writing a wellscoped method that checks in the ebvironment of local+global variables to see whether our input is well scoped or not.

```
def wellScoped(program)
        environment = program.globals
        helper(program.stmt, environment)
    end

    def helper(prog, env)
        case prog
        when GCSkip
            true
        when GCLocal
            helper(prog.stmt, env.push(prog.var))
        when GCAssign
            env.include?(prog.var) && helper(prog.expr, env)
        when GCCompose
            helper(prog.val1, env) && helper(prog.val2, env)
        when GCIf
            prog.each do |guard|
                helper(guard[0], env) && helper(guard[1], env)
            end
        when GCDo
            prog.each do |guard|
                helper(guard[0], env) && helper(guard[1], env)
            end
        when GCComp
            helper(prog.val1, env) && helper(prog.val2, env)
        when GCAnd
            helper(prog.val1, env) && helper(prog.val2, env)
        when GCOr
            helper(prog.val1, env) && helper(prog.val2, env)
        when GCTrue
            true
        when GCFalse
            true
        when GCConst
            true
        when GCVar
            env.include?(prog.val)
        when GCOp
            helper(prog.val1, env) && helper(prog.val2, env)
        end
    end
```

We start with a list of global variables and work our way through each case by checking its arguments for well scoped. If we encounter a local variable, we reset this local variables for the purpose of checking the scope of its arguments, we then reset it to its original value after we are done so as to not clash with a possible global variable of the same name.
We then define an eval method to evaluate our input and return the memory state that maps variable to integers. We take the help of a reduce function that evaluates each term using the memory state (this is the big step of the semantics) to help us with simpler evaluation.

```ruby
def eval(program)
        environment = Hash.new do |hash, key|
            raise("Accessing a variable that doesn't exist!")
        end
        program.globals.each do |guard|
            environment[guard] = "something"
        end
        end_env = helper2(program.stmt, environment)
    end

    def helper2(prog, env)
        case prog
        when GCSkip
            return env
        when GCAssign
            variable = prog.var
            value = reduce(prog.expr, env)
            env[variable] = value
            return env
        when GCCompose
            res1 = helper2(prog.val1, env)
            helper2(prog.val2, res1)
        when GCLocal
            if env.include?(prog.var)
                tmp = env[prog.var]
                env[prog.var] = "no_value"
                newenv = helper2(prog.stmt, env)
                newenv[prog.var] = tmp
                return newenv
            else
                env[prog.var] = "no_value"
                newenv = helper2(prog.stmt, env)
                newenv.delete(prog.var)
                return newenv
            end
        when GCIf
            trueguards = []
            allguards = prog.guards
            allguards.each do |guard|
                if reduce(guard[0], env) == true
                    trueguards.push(guard[1])
                end
            end
            if trueguards.length() > 0
                helper2(trueguards.sample, env)
            else
                return env
            end
        when GCDo
            trueguards = []
            allguards = prog.guards
            allguards.each do |guard|
                if reduce(guard[0], env) == true
                    trueguards.push(guard[1])
```

```
                end
            end
            if trueguards.length() > 0
                newenv = helper2(trueguards.sample, env)
                recreate = GCDo.new(allguards)
                return helper2(recreate, newenv)
            else
                return env
            end
        end
    end

    def reduce(prog, env)
        case prog
        when GCConst
            prog.val
        when GCVar
            env[prog.val]
        when GCOp
            case prog.op
            when :plus
                reduce(prog.val1, env) + reduce(prog.val2, env)
            when :minus
                reduce(prog.val1, env) - reduce(prog.val2, env)
            when :times
                reduce(prog.val1, env) * reduce(prog.val2, env)
            when :div
                reduce(prog.val1, env) / reduce(prog.val2, env)
            end
        when GCComp
            case prog.op
            when :eq
                reduce(prog.val1, env) == reduce(prog.val2, env)
            when :less
                reduce(prog.val1, env) < reduce(prog.val2, env)
            when :greater
                reduce(prog.val1, env) > reduce(prog.val2, env)
            end
        when GCAnd
            reduce(prog.val1, env) && reduce(prog.val2, env)
        when GCOr
            reduce(prog.val1, env) || reduce(prog.val2, env)
        when GCTrue
            true
        when GCFalse
            false
        end
    end
```

The working of the method is very similar to previous functions we have defined with the only tricky case being the existence of a local variable with the same symbol as a global variable. I have defined the method to prevent clashes and incorrect overrides to occur.