Advanced Programming, a.y. 2016/2017
Homework 1
Student: Orlando Leombruni, *matricola* 475727

Quick start

The domain-specific language and its interpreter are implemented in the module Dsl, contained in the "src/lib/dsl.ml" file; it is treated as a library. The network modelization is in the file "src/network.ml", while a simulation of the behavior of a network is in the file "src/test.ml". All .ml files are fully documented via code comments, which illustrate signatures for the various functions and concise comments on design choices.
Provided with the source files is a Makefile, which uses ocamlbuild to fully automatize the compilation process. The available commands are

- make native: compiles the simulation into a native (binary) code program
- make byte: compiles the simulation into a bytecode program
- make lib: compiles the domain-specific language and its interpreter into a library (found inside the /lib directory)
- make test: compiles (into native code) and immediately executes the simulation
- make all: compiles the library and the simulation, in both native- and byte-code
- make clean: removes temporary files and all previous compilation outputs.

The Domain-Specific Language

The following is a short introduction on the syntax and semantics of the DSL, analyzing every kind of expression of the language and its possible value after the evaluation.

- **Filter f**, where f is a *function from packets to booleans*
  Evaluates to itself. It represents the activation of a filter, which is an OCaml function from packets to booleans; if the packet satisfies the predicate of the filter, it is NOT forwarded.
- **Interface i**, where i is a list of integers
  Evaluates to itself. It is the controller's representation of the output interfaces of a switch, used in other expressions.
- **Network i**, where i is a list of integers
  Evaluates to itself. It is the controller's representation of the network, used in other expressions.
- **Rule (src, dst, outlink, active, max_read, max_write)**
  - where: src, dst, outlink, max_read, max_write are integers, active is a boolean

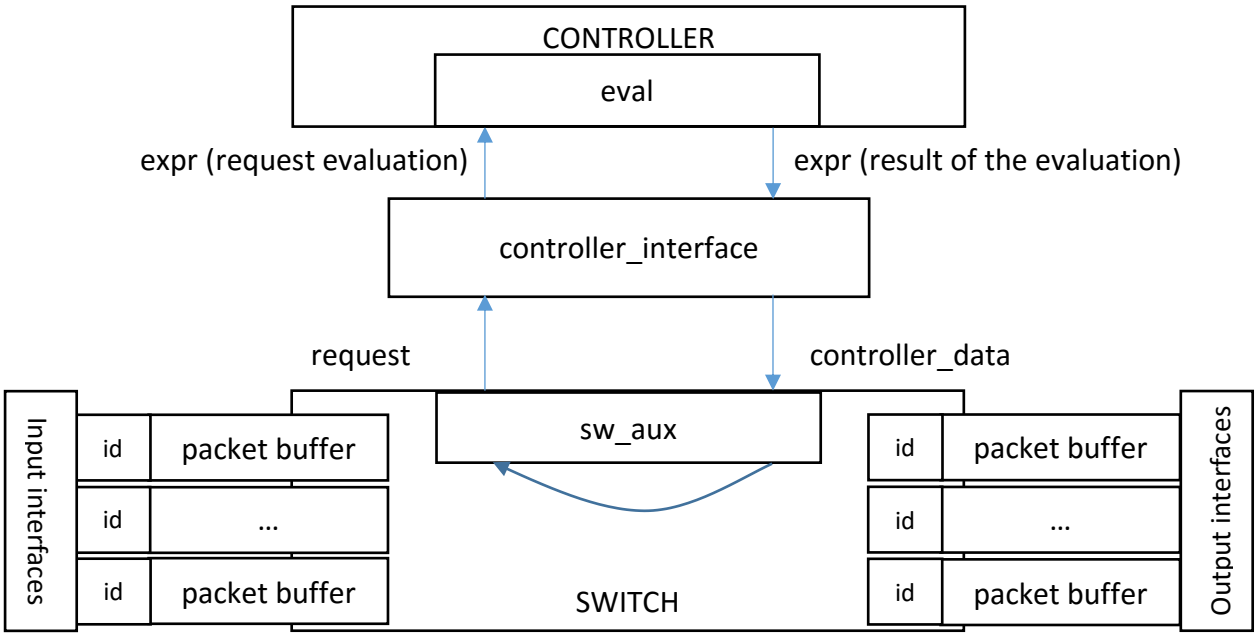
  Evaluates to itself. It is the controller's representation of a forwarding rule, together with the parameters used in defining security policies. If such security policies are not active, max_read and max_write will be set to -2 (a conventional unreachable value)
- **RoutingTable rt**, where rt is a (OCaml) list of expressions
  Evaluates to itself after checking that rt's elements are Rules. The controller's representation of a routing table.
- **GetRules (i, e1, e2)**, where i is an integer and e1 and e2 are expressions
  Evaluates to a routing table after evaluating e1 into an Interface and e2 into a Network. The resulting routing table contains n rules having i as the destination address, one for every element of e2 (as the source address). For simplicity, every rule has the first output interface of e1 as outlink.

- **GetRulesWithSP (i, e1, e2, (r,w))**, where i, r and w are integers and e1 and e2 are expressions
  Same as the previous one, but puts the values for r and w into the security policy fields of the generated rules.
- **Install(e1, e2)**, where e1 and e2 are expressions
  Evaluates to a routing table after evaluating e1 into an Interface and e2 into a Network. The routing table is built in this way: let n be the number of output interfaces and m be the number of addresses in the network. Then, for every element of e2 as the source address, compute m rules; the i-th rule will have the i-th element of e2 as destination address and the (i mod n)-th element of e1 as outlink. As a minimal example, take e1 = [1,2], e2 = [1,2,3]; Install will create the following routing table (ignoring irrelevant fields of Rules):

Source	Destination	Outlink
1	1	1
1	2	2
1	3	1
2	1	1
2	2	2
2	3	1
3	1	1
3	2	2
3	3	1

- **InstallWithSP(r,w,e1,e2)**, where r and w are integers and e1 and e2 are expressions
  Same as the previous one, but puts the values for r and w into the security policy fields of the generated rules.
- **Drop(e1,e2)**, where e1 and e2 are expressions
  Evaluates to a routing table after evaluating e1 into a Rule and e2 into a RoutingTable. If e1 is contained in e2, returns a RoutingTable equal to e2 with the “active” field of the rule corresponding to e1 set to false.
- **Undrop(e1,e2)**, where e1 and e2 are expressions
  Same as the previous one, but sets the “active” field of the abovementioned rule to true.
- **Delete(e1,e2)**, where e1 and e2 are expressions
  Evaluates to a routing table after evaluating e1 into a Rule and e2 into a RoutingTable. If e1 is contained in e2, returns a RoutingTable equal to e2 but without the rule corresponding to e1, else returns e2 itself.
- **Redirect(from,to,e1)**, where from and to are integers and e1 is an expression
  Evaluates to a routing table after evaluating e1 into a RoutingTable. It searches e1 for a rule with “to” as the destination address, takes the outlink value of this rule (call it “o”), then goes into e1, replacing rules having “from” as the destination address with equivalent ones having “o” as the outlink.

The Network Modelization



Further information about the modelization is provided in form of code comments inside the "src/network.ml" file.

Simulation

The file "src/test.ml" contains a simulation of a network. 25 packets are pre-loaded in the switch's input interfaces and are routed according to the rules and updates given by the controller via the controller_interface function; the process is mostly invisible, save for a few information about dropped packets and requested updates. An assert command at the end verifies that the output is the desired one. More information, including a detailed step-by-step walkthrough of the simulation, is included in the aforementioned file via comments.

Optional Assignment

Security Policies

Security policies in this implementation can be of three types: limit the number of "read", "write" or both operations between a source and a destination. Policies are global, i.e. applied to every src/dest pair. The remaining number of read and write operations is maintained inside the switch's representation of a forwarding rule; in order to avoid unnecessary duplications of the whole forwarding table (which is an immutable data structure), these counters are mutable fields. When the policy "limit to n reads, limit to m writes" becomes active, all rules set their corresponding fields to n and m; every time the switch processes a packet, it reads its payload in order to find the type of operation, and then check if the corresponding counter is greater than zero. If the condition is satisfied, the packed is router and the counter decremented, else the packet is dropped. A value of -2 in either one of the counters means that an unlimited number of the corresponding operation is permitted; in fact, if there's no active policy, both counters are set to -2.

Filters

Filters are implemented as (OCaml) functions from packet to booleans; this gives the programmer maximum freedom of choice. If a packet satisfies the conditions of a filter, it is dropped.

Mobility

Mobility is handled by modifying the routing table in order to route packets intended to the old address into the output link associated with the new address. More info can be found in the DSL section.

Concluding remarks

The code has been written using Sublime Text and executed/compiled with OCaml 4.03.0, installed via Homebrew on macOS Sierra 10.12.1. While I wrote the entirety of my code, the solution has been designed and debated with the help of fellow students Giuseppe Astuti, Alessandra Fais and Davide Scano.