Report assignment homework 2

Carlo Leo 546155

As the kernel of the functional language was considered the presented functional language together with four abstract operations on socket (like in homework1), over which was developed the possibility of defining security policies. Each operation is mapped into a symbol as shown in the following table:

Operation	**Symbol**
OpenSocket	‘o’
SendData	‘w’
ReadData	‘r’
CloseSocket	‘c’

Together forms the global alphabet which can be used to define a policy.
Policies are represented as deterministic finite automata such that every state is being considered as accepting one, so a string is not accepted when the transaction function is undefined for a given couple *(symbol, state)*. Representing execution history as a string (*type history*), this approach is natural to verify a policy. A data structure, *type policyStack*, was defined to keep track of all active policies during program execution. Moreover, two policies are provided by language:

- No SendData after ReadData *(pNwAr)*
- No a CloseSocket before before a OpenSocket (*pNcBo)*

For details see file *policy.ml* and *dfa.ml*.
To make possible the definition of security policies to programmer two instructions have been added to the language:

- BuildPolicy
- DefPolicyIn

The first one takes as parameters the list states, initial state and transition function as list of possible transitions (*type transition)*, it verifies parameters (e.g. if transitions are defined over legal symbols/states) and returns a value, which represents a policy at run time. To do so, expressible values (*type envT*) were extended by *Policy* constructor.
The second one, DefPolicyIn, was introduced to bind a snippet of code by a policy. Indeed, it takes two expressions as parameters: the first one, once evaluated, must match to the policy while the second one corresponds to the code. In this way a programmer can define, besides **local policies**, **global** and **nested policies** as well.
In order to check all active policies over the global execution history, the interpreter delegates execution of the program to the function *pval*, which expects as parameters, besides the expression and environment, a parameter of type *policyStack* and one of type *history*. Unlike eval, it returns a pair *(envT,history*), so

eval has to match its return value to return only the computed value *envT*. That because, peval checks active policies each time it should execute an relevant operation (see table above) in two step:

- Extending history with current operation (calling *updateH*)
- Verifying that resulting history is accepted by each active policy (calling *checkPolicy*)

If the checking succeeds then execution goes on, propagating the extended global execution history, otherwise execution is being aborted. In this way, we are sure that all policies were obeyed, so eval can return the value.
As requested the language was equipped with recursive functions as well, introducing a constructor, *LetRec*, to define them and a constructor to represent them at run time extending expressible values by *RecFunVal*. Since, unlike the lamba, we also need to keep a binding between function name and its closure in order to replace its name occurrences within its body correctly.