Language-based Technology for Security

Assignment #2

Antonio Strippoli, Giulio Paparelli

03/05/2021

For the realization of the project it was decided to reuse the **Ocaml** code that we have seen in class as a skeleton. The code also includes the **DFA** implementation to represent the security policies, adapting it according to the rest of the interpreter implementation. It was decided to not implement a check on the correctness of the automaton, as it is done at runtime when verifying a policy (from the **dfa_accepts** function).

We have extended the functional language to support the following expressions:

- **EChar**: to represent a character;
- **ETransition**: to represent a $< state, symbol, state >$ transition of a DFA;
- **ESingleList, EList**: to represent a list of variable size. **ESingleList** represents a list of only one element. **EList** represents a list of 2 or more elements, according to the grammar $EList := (expr * EList) \mid (expr * expr)$;
- **Let, LetIn**: to represent Ocaml's *let* and *let-in* commands;
- **FunR, Letrec**: to represent the definition and evaluation of a recursive function;
- **CSeq**: to represent a sequence of commands, necessary to demonstrate how the verification of policies does not occur only on the elements of the stack;
- **Read, Write**: to represent privileged (fictitious) operations whose execution is permitted based on the policies in use;
- **Policy** to represent an automaton that describes a security property (eg *Chinese Wall*);
- **Phi** to represent the control of a policy for an expression.

Consequently, the **value** type has also been extended to represent characters, transitions, lists, (closures of) recursive functions and DFAs.

Note that thanks to the implementation of the list and transition types, it is possible to build a **abstract syntax tree** completely derived from the implemented functional language (i.e. not making use of Ocaml's internal types). This was found to be essential for the programmer to define custom **Policy**.

We added 2 parameters to the **eval** function that implements the interpreter:

- **eta** (*string*): to represent the history of privileged operations already performed;
- **dfa_list** (*DFA list*): to represent the list of security policies that must be respected.

Finally, we have **tested the interpreter** for 2 categories of cases:

- `op1 ; Phi (policy, op2)` where op_i is a privileged operation and *policy* is a security policy between {`no-read-after-write`, `no-write-after-read`};
- `op1 ; Phi (noRaW, Phi (noWaR, op2))` where $op\ _{texti}$ is a privileged operation, $noRaW$ is the policy `no-read-after-write` and $noWaR$ is the `no-write-after-read` policy.
  This means that `op2` can only be the same type of operation as `op1` (e.g. op_1 = Read, op_2 = Read).