Homework 1 report

Language Based Technology for Security

Niccolò Piazzesi
n.piazzesi@studenti.unipi.it

April 26, 2022

1 Introduction

In this report i will describe the general design and implementation choices of the simple functional and mobile language assigned as homework. I will describe the main features and its semantics, especially how i have handled the security aspect of mobile code. In the final section i will also describe some other features that could further extend the power of the language.

1.1 Running the tests and examples

To build and test the code, you must have **Dune** and **QCheck** installed on your system. This can be done with the following command:

```
$ opam install dune qcheck
```

The following commands can be executed from the root of the project:

- Running tests

  ```
  $ dune test -f
  # -f makes the tests run even if the target was already built
  ```

- Running the simple examples

  ```
  $ dune exec hw1
  ```

2 The language

2.1 Syntax

The language is an extended version of the λ-calculus, equipped
with other linguistic construct to ease its use. It features integer and boolean literals, arithmetic and logic binary expressions, variables and variable bindings, anonymous functions, function application, and a special form of binding to

handle named (and possibly recursive) functions. The other primitive construct is **execute** which handles the sandboxed-execution of remote code. The abstract syntax of the language can be described with the following formal grammar:

$$\text{Aexp} := \text{n} \in \mathbb{N} \mid \textbf{Exp} \text{ aop } \textbf{Eexp} \quad \text{aop} \in \{+, -, *, /\}$$

$$\text{Bexp} := \text{b} \in \{\text{true, false}\} \mid \textbf{Exp} \text{ bop } \textbf{Exp} \quad \text{bop} \in \{<, >, =\}$$

$$\begin{aligned} Exp := {} & \text{Aexp} \\ & \mid \text{Bexp} \\ & \mid x, y, \ldots \\ & \mid \textbf{if } Exp \textbf{ then } \ Exp \ \textbf{ else } Exp \\ & \mid \textbf{let } x = Exp \textbf{ in } Exp \\ & \mid \textbf{fun } x -> Exp \\ & \mid \textbf{let fun } f\ x = Exp \textbf{ in } Exp \text{ (*function definition*)} \\ & \mid Exp\ Exp \text{ (* function application *)} \\ & \mid \textbf{execute } Exp \textbf{ allowing } [p \textbf{ s.t } p \in \{\text{access(x),execute,binary\_ops}\}] \end{aligned}$$

To give an example, a possible code to calculate the factorial of 6 could look like this:

```
let fun fact n =
    if n = 0 then
        1
    else
        fact n*fact n-1
in
fact 6
```

In the concrete language one may imagine to have other higher level linguistic constructs, but for our purpose the syntax shown is more than enough.

2.2 Basic semantics

Values Each expression gets interpreted to three possible types of value:

1. An integer.
2. A boolean.
3. A closure value. Closures represent the runtime value of functions, and they keep all the information necessary to access and apply such functions.

Environment To handle variable binding and access we need a data structure to track all the mappings. This is done inside the **symbol table**.

A symbol table is an hash table using variable names as key to index the variable runtime value. In the implementation the environment is actually made of a list of symbol tables. This is done to handle scoping and variable hiding inside functions.

Scoping Scoping is **lexical** and the head of the list represent the innermost scope. When an expression requires to access a variable, the interpreter scans the list left to right until it finds the first table containing a valid mapping, and returns the value found in that table. Each function body is evaluated inside a new scope.

2.2.1 Arithmetic and boolean expressions

The semantics of arithmetic and boolean expressions are defined inductively as expected. All the operators take integer operands. Division is interpreted as **integer** division, meaning that we take the quotient and discard the remainder.

2.2.2 General basic expressions

If conditional If expression are defined as usual. If the guard condition is true the interpreter evaluates the **then** branch gets evaluated otherwise it goes to the **else** branch.

Let bindings The interpreter updates the environment with the mapping **x, exp1** and then evaluates the **in** expression.

Function declaration Anonymous functions simply gets evaluated to their closure, saving the parameter name and capturing the environment.

Named functions gets evaluated to their closure as well and the mapping **(f, closure(f))** is saved before evaluating the **in** expression.

Using let bindings, we can also bind a name to an anonymous functions, but there is a big difference with primitive named functions. In the latter case, the mapping **(f, closure(f))** is captured in the environment of the function itself, meaning that we allow recursive function definition, as seen in the factorial basic example. For anonymous function this is obviously not possible.

Function application Function application is defined as usual. The left expression gets evaluated. If it is a closure of a function we apply it passing the evaluated right expression as parameter, producing an error whenever we try to apply any other expression.

2.3 Execute semantics

As expected, the most complex part of the language is the handling of mobile code. This is done using the primitive **execute**. Execute takes two parameters: the mobile code to be evaluated and the list of permissions, which represent actions allowed inside the mobile code.

The list of permission is specified by the receiving end of the remote code and can be of three types:

1. An **access(x)** permission allows the mobile code to access name x from the surrounding code. This can be used to allow access to private data when x is a variable, but also to allow calling of function x.

2. A **binary_ops** permission allows the mobile code to perform arithmetic and boolean expression. One could decide to disable these actions when the lack of control on the data used could cause security issues (e.g an overflow caused by summing two big integers).

3. The **execute** permission allows the execute expression to be called from another execute in a restricted way. This should be used very carefully, and as i will explain later is handled in a restrictive way, to prevent security problems. A completely valid alternative would obviously be to never allow nested execution.

As an example, let's say that the mobile code computes the factorial using the user defined function. The corresponding abstract expression would look like this:

```
execute (fact 5) allowing [access("fact"), binary_ops]
```

By default, **no** possibly insecure action is allowed.

The access permission models in an abstract way more concrete permissions. In a concrete language you would have, for example, filesystem read and write permissions and network communication permissions such as **send** to allow the communication of private values. In our simple example, these can all be modelled as permissions to access certain function which could be called "read_file", "write_file" or "send".

2.3.1 Sandbox execution

Mobile code gets executed by the interpreter in a special restricted environment, **a sandbox**. A sandbox is made of three parts:

1. The list of permissions specified by the receiving code.

2. A reference to the external environment, containing all the name mapping up until the current moment.

3. A new internal environment, used to store all the new mapping introduced by the mobile code.

The internal environment is completely fresh and empty at the beginning,not sharing any part with the external one. Evaluation proceeds inside this sandbox, and we performs the following checks on security relevant expressions:

1. For variable access, we first check if it exists in the internal environment. If that is not the case, the interpreter checks that access of that name is allowed inside the sandbox. If it is, it proceeds to search the variable mapping in the external environment. If it is not, it raises a security violation.

2. For binary expressions, we check that arithmetic operations are allowed before evaluating the operands. If not, a security violation is raised.
3. When an execute expression tries to use another execute, we first check that nested executes are allowed. If they are, the interpreter evaluates the new execute inside a new sandbox. This new sandbox is instantiated with an empty internal environment. The permissions are taken from the first original execute, ignoring the permissions set in the mobile code calling the nested one. This prevents the overwriting of original permissions, avoiding the introduction of possible private data leakage.

Allowing nested executions could case other issues, such as non-termination (think of a cyclic execute where some code calls some other remote code that calls the original code and so on...) but i believe that the presented semantics allows them in a controlled and safer way. As said before, one could argue that the best possible choice is to disable nested executes completely, but i feel that is interesting to think of safe ways to introduce such constructs.

3 Testing

The implementation has been checked by using property based testing and the QCheck library. More specifically, i verified the security relevant aspect of the interpreter, which meant verifying two relevant properties:

1. **Every insecure expression raises a security violation error.**
2. **Secure expressions never raise a security violation error.**

3.1 Generating test cases

The more challenging part of testing was handling the generation of random expression as test cases. Using uncontrolled randomness would result in many useless test, as the majority of possible generated expressions would make no computational sense (e.g applying an integer as a function or adding two lambdas). To generate useful tests i have used a type driven approach, which is popular in testing language implementations [1]. Every valid expression is assigned one of three possible types:

1. Int
2. Bool
3. Fun(t1,t2)

where Fun(t1,t2) represents a function from type t1 to t2. Handling variable typing was done using a typing context Γ which is a list of mapping **(x,t)**, where **t** represent the type of variable **x**. The expression generator takes a type T as input and generates a valid expression of type T. As an example, if we want to generate an integer expression, the generator chooses at random between a integer literal, a binary operation between two other integer expressions and resulting in an integer or a composite expression (let,letfun, if, or function call) that recursively generate valid sub expressions and evaluates to an integer. For simplicity the basic generator does not generate execute expressions, which are handled differently to generate insecure expressions.

3.2 Generating insecure expressions

Generating insecure expressions involves two things:

1. Generating a execute which evaluates a possibly security relevant action **A**.
2. Disabling action **A** inside the sandbox

The execute generator takes a random relevant action A to check and produces one of the following thing:

- If A is a **access(x)** it simply creates a variable access **x**.
- If A is **execute** it produces a new execute with a random nested expression.
- If A is **binary_ops** it creates a random binary expression.

To ensure that the expression in insecure we create a random list of permissions and remove the generated action A from it.

The generator does not create a composite expression. While this restricts the cases actually tested, i think that generating more complex insecure expressions adds a lot of complexity to the general expression generator without resulting in many more relevant cases, because in these cases we must always end up in one of the generated cases as sub expressions.

4 Possible extensions

The language presented is a very simple and minimal one. One could think of various linguistic constructs to enrich it and make it more powerful. Other than more syntactic extensions, i think of two possible extensions, that could make it more safe and secure:

- **Code signature**. Mobile code could be signed with a cryptographic key, and an interpreter could reject code that it does not trust entirely.
- **Type system**. A very basic type system could be added to the language itself, which would prevent many wrong expressions to be evaluated.

A more advanced solution,based on type systems (and formal logic in general), but out of scope for our simple example, could be **proof carrying code**[2].

The basic idea is the following: the host system can determine if it safe to execute untrusted code. To do this, the untrusted code must also send a *proof* that states that the code satisfies a certain security policy specified by the host system itself. The system can then check that the proof is valid before executing the code.

References

[1] Pedro Vasconcelos. Verifying a simple compiler using property-based random testing.

[2] George C Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.