Language Definition

Niccolò Piazzesi
n.piazzesi@studenti.unipi.it
Language Based Technology for Security
Homework 1 report

April 28, 2022

In this report i will describe the general design 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.

1 The language

1.1 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:

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

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

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

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.

1.2 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.

1.2.1 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.

1.2.2 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.

1.3 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.

By default, **no** permission is set.

1.3.1 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 potential issues.

2 Some examples

- Computing factorial of 6

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

- Valid mobile code execution accessing an external function

```
let fun send x =
    #actual code to send a value
    0
in
let x = 12345 in
execute send(x)  allowing [access("send")]
```

- Some invalid executes

  - Invalid name access

```
let secret_val = 34 in
execute (fun x -> x+3) secret_val
allowing [binary_ops]
```

  - No arithmetic allowed

```
let secret_val = 34 in
execute 3*5-10/2
allowing []
```

– No nested execution

```
let mySum = (fun x -> (fun y -> x+y)) in
let aValue = 6 in
execute
    let bValue = 5 in
    execute
        mySum aValue bValue
    allowing []
allowing
[binary_ops, access("aValue"), access("bValue")]
```