LBTS project report – ScaML[2]

Andrea Simone Costa Alessio Duè Alessandro Scala

June 27, 2024

1 Syntax

The syntax of ScaML[2] is inspired by OCaml. Some notable differences include:

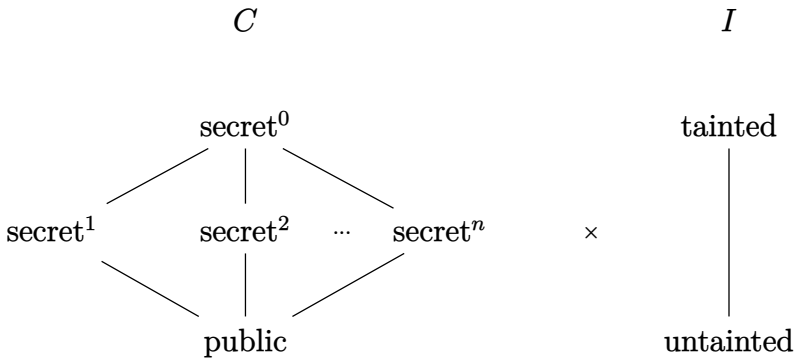
- `let` bindings accept a list of attributes, like `public` or `tainted`;
- operators for endorsing and declassifying values;
- assertions and predicates on attributes of values;
- plugins, with dynamically checked interfaces.

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid e.x \mid \texttt{(}e\texttt{)} \\
& \mid & e\ \mathit{bop}\ e \mid \mathit{uop}\ e \mid \texttt{[}e\texttt{,}\ldots\texttt{,}e\texttt{]} \mid e\texttt{[}e\texttt{]} \mid \texttt{length}\ e \\
& \mid & e\ e \mid \lambda x.\ e \mid \texttt{fix}\ e \mid \texttt{fix*}\ e \mid \mathit{let} \\
& \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \texttt{with}\ h\ \texttt{handle}\ e \mid \texttt{do}\ \mathit{op}\ e* \\
& \mid & \texttt{module}\ d*\ \texttt{end} \mid \texttt{trusted module}\ d*\ \texttt{end} \mid \texttt{plugin}\ s\ \mathit{intf}*\ \texttt{end} \\
& \mid & \texttt{die} \mid \texttt{assert}\ e \mid \texttt{has\_attr}\ \mathit{attr}\ e \\
& \mid & \texttt{declassify}\ e \mid \texttt{endorse}\ e \mid \texttt{declassify\_pc}\ e \mid \texttt{endorse\_pc}\ e \\
& \mid & \texttt{print}\ e \mid \texttt{get} \\
\\
\mathit{let} & ::= & \texttt{let}\ \mathit{attr}*\ x\ \texttt{=}\ e\ \texttt{in}\ e \\
& \mid & \texttt{let rec}\ \mathit{attr}*\ x = e\ (\texttt{and}\ \mathit{attr}*\ x = e)*\ \texttt{in}\ e \\
\\
\mathit{intf} & ::= & x\ \texttt{:}\ \tau \\
d & ::= & \texttt{let}\ \mathit{attr}*\ x = e \\
& \mid & \texttt{let rec}\ \mathit{attr}*\ x = e\ (\texttt{and}\ d)* \\
& \mid & \texttt{export}\ x \\
\\
h & ::= & \texttt{\{}(\mathit{op}\ x\texttt{+=}\ e\texttt{,})*\ \texttt{return}\ x = e\texttt{\}} \\
\\
v & \in & \mathrm{Int} \cup \mathrm{Str} \cup \mathrm{Bool} \\
s & \in & \mathrm{Str} \\
x, \mathit{op} & \in & \mathrm{Ide} \\
\\
\mathit{attr} & ::= & \texttt{public} \mid \texttt{secret} \mid \texttt{tainted} \mid \texttt{untainted} \\
\tau & ::= & \texttt{any} \mid \texttt{int} \mid \texttt{string} \mid \texttt{bool} \mid \tau\ \texttt{->}\ \tau \mid \texttt{[}\tau\texttt{,}\ldots\texttt{,}\tau\texttt{]} \mid \texttt{(}\tau\texttt{)}
\end{array}
$$

[2]From the names of its authors

2 Operational semantics

Values are tagged with labels, which belong to a lattice Lbl. Lbl is itself the product of the confidentiality and integrity lattices, I and C, defined by the following Hasse diagrams:



secret^p is the confidentiality label of a secret value created in the p-th plugin; in particular $p = 0$ is used for the modules defined in the main program file.

We define the big-step operational semantics relation, with signature:

$$\Downarrow : \text{Env} \times \underbrace{\text{Lbl}}_{pc} \times \text{Exp} \to \text{Val} \cup \underbrace{(\text{Ide} \times \text{Fun} \times \text{Val}^* \times \text{Lbl})}_{\text{for raising operations}}$$

where:

$$\begin{aligned} \text{Val} &= \text{BareVal} \times \text{Lbl} \\ \text{BareVal} &= \text{Int} \cup \text{Str} \cup \text{Bool} \cup \text{Fun} \cup \underbrace{\text{Val}^*}_{\text{tuples}} \cup \text{Defer} \cup \text{Mod} \cup \underbrace{\text{ValType}}_{\text{value with interface}} \end{aligned}$$

$$\begin{aligned} \text{Int} &= \mathbb{N} \\ \text{Str} &= \text{ASCII}^* \\ \text{Bool} &= \{\mathbf{tt}, \mathbf{ff}\} \end{aligned}$$

$$\begin{aligned} \text{Env} &= \text{Val}^{\text{Ide} \cup \{\text{@trusted,@plugin}\}} \\ \text{Fun} &= \left\{ \lambda^{\Delta} x.\ e \mid \Delta \in \text{Env}, x \in \text{Ide}, e \in \text{Exp} \right\} \\ \text{Defer} &= \left\{ \text{defer}^{\Delta}\ e \mid \Delta \in \text{Env}, e \in \text{Exp} \right\} \end{aligned}$$

$$\text{Mod} = \{\text{mod}\ \Delta \mid \Delta \in \text{Env}\} \cup \{\text{tmod}\ \Delta \mid \Delta \in \text{Env}\} \cup \{\text{plug}\ \Delta \mid \Delta \in \text{Env}\}$$

$$\text{ValType} = \{v : \tau \mid v \in \text{BareVal}, \tau \in \text{Type}\}$$

and Exp is the set of syntactic elements generated by the nonterminal e in the grammar, and Type by the nonterminal τ.

Defer values are needed to implement the fixpoint operators `fix` and `fix*`, they cannot be created by the user. Modules are environments tagged with the module

kind: simple module, trusted module or plugin.[3] Modules are program units used to group together code relative to a functionality. Since each module has its own environment and only exposes names declared with `export`, they are useful to provide encapsulation, giving complete control to the programmer about what is exposed and what is hidden. Every program file implicitly defines a module, and therefore has to follow module syntax. @trusted and @plugin are special identifiers used to keep track of the trustedness level and plugin identifier in environments.

Following are the inference rules. We begin with simple rules for values, variables and tuple projections and length:

$$\frac{}{\Delta, pc \vdash v \Downarrow \langle [\![v]\!],\ pc\rangle}\ [\text{Val}]$$

$$\frac{}{\Delta, pc \vdash \lambda x.\ e \Downarrow \langle \lambda^{\Delta} x.\ e,\ pc\rangle}\ [\text{Lam}]$$

$$\frac{\forall i \in [1,n].\ \Delta, pc \vdash e_i \Downarrow \langle v_i,\ \ell_i\rangle \triangleq \mathbf{v}_i}{\Delta, pc \vdash \texttt{[}e_1\texttt{,}\ \ldots\ \texttt{,}e_n\texttt{]} \Downarrow \langle [\mathbf{v}_1, \ldots, \mathbf{v}_n],\ pc \sqcup \bigsqcup_{i=1}^{n} \ell_i\rangle}\ [\text{Tuple}]$$

$$\frac{\Delta(x) = \langle v, \ell\rangle \quad v \neq \text{defer}^{\Delta'}\ e}{\Delta, pc \vdash x \Downarrow \langle v,\ pc \sqcup \ell\rangle}\ [\text{Var}] \qquad \frac{\Delta(x) = \langle \text{defer}^{\Delta'}\ e, \ell_1\rangle \quad \Delta', pc \vdash e \Downarrow \langle v,\ \ell_2\rangle}{\Delta, pc \vdash x \Downarrow \langle v,\ pc \sqcup \ell_1 \sqcup \ell_2\rangle}\ [\text{VarDefer}]$$

$$\frac{\begin{array}{c}\Delta, pc \vdash e_1 \Downarrow \langle [\langle v_j, \ell_{v_j}\rangle]_{j\in[0,\ldots,n-1]},\ \ell_1\rangle \\ \Delta, pc \vdash e_2 \Downarrow \langle i,\ \ell_2\rangle \quad 0 \leq i \leq n-1 \\ v_i \neq \text{defer}^{\Delta'}\ e\end{array}}{\Delta, pc \vdash e_1\texttt{[}e_2\texttt{]} \Downarrow \langle v_i,\ pc \sqcup \ell_1 \sqcup \ell_2 \sqcup \ell_{v_i}\rangle}\ [\text{Proj}]$$

$$\frac{\begin{array}{c}\Delta, pc \vdash e_1 \Downarrow \langle [\langle v_j, \ell_{v_j}\rangle]_{j\in[0,\ldots,n-1]},\ \ell_1\rangle \\ \Delta, pc \vdash e_2 \Downarrow \langle i,\ \ell_2\rangle \quad 0 \leq i \leq n-1 \\ v_i = \text{defer}^{\Delta'}\ e \quad \Delta', pc \vdash e \Downarrow \langle v,\ \ell_3\rangle\end{array}}{\Delta, pc \vdash e_1\texttt{[}e_2\texttt{]} \Downarrow \langle v,\ pc \sqcup \ell_1 \sqcup \ell_2 \sqcup \ell_{v_i} \sqcup \ell_3\rangle}\ [\text{ProjDefer}]$$

$$\frac{\Delta, pc \vdash e \Downarrow \langle [\mathbf{v}_j]_{j\in[0,\ldots,n-1]},\ \ell\rangle}{\Delta, pc \vdash \texttt{length}\ e \Downarrow \langle n,\ pc \sqcup \ell\rangle}\ [\text{Length}]$$

The idea of deferred expressions is that $\text{defer}^{\Delta}\ e$ automatically evaluates e in environment Δ when the defer is extracted from the environment or a tuple field.

[3] Actually, this distinction is not strictly needed for the semantics of (trusted) modules and plugins, but we keep it in order to be able to print them differently.

Then we have rules for application:

$$\frac{\Delta, pc \vdash e_1 \Downarrow \langle \lambda^{\Delta'} x.\ e,\ \ell_f \rangle \quad \Delta, pc \vdash e_2 \Downarrow \mathbf{v} \quad \Delta'[x \mapsto \mathbf{v}], pc \sqcup \ell_f \vdash e \Downarrow \langle u,\ \ell_u \rangle}{\Delta, pc \vdash e_1\ e_2 \Downarrow \langle u,\ pc \sqcup \ell_f \sqcup \ell_u \rangle}\ [\text{App}]$$

$$\frac{\begin{array}{c}\Delta, pc \vdash e_1 \Downarrow \langle (\lambda^{\Delta'} x.\ e) : \tau_1 \to \tau_2,\ \ell_f \rangle \quad \Delta, pc \vdash e_2 \Downarrow \mathbf{v} \quad \mathbf{v}' \triangleq \text{apply-intf}(\mathbf{v}, \tau_1) \\ \Delta'[x \mapsto \mathbf{v}'], pc \sqcup \ell_f \vdash e \Downarrow \langle u,\ \ell_u \rangle \triangleq \mathbf{u} \quad \langle u', \ell'_u \rangle \triangleq \text{apply-intf}(\mathbf{u}, \tau_2)\end{array}}{\Delta, pc \vdash e_1\ e_2 \Downarrow \langle u',\ pc \sqcup \ell_f \sqcup \ell'_u \rangle}\ [\text{AppIntf}]$$

Application is standard if e_1 evaluates to a closure. If instead it evaluates to a closure tagged with a type $\tau_1 \to \tau_2$ (its interface), then we need to check that the parameter and the result match τ_1 and τ_2. We do so with the help of the following function:

$$\text{apply-intf}(\langle v, \ell \rangle, \tau) = \begin{cases} \langle v, \ell' \rangle & \begin{array}{r}\tau = \texttt{any} \\ \vee\ (\tau = \texttt{int} \wedge v \in \text{Int}) \\ \vee\ (\tau = \texttt{string} \wedge v \in \text{Str}) \\ \vee\ (\tau = \texttt{bool} \wedge v \in \text{Bool})\end{array} \\ \langle v : \tau, \ell' \rangle & \tau = \tau_1 \to \tau_2 \\ \langle [\text{apply-intf}(\mathbf{v}_1, \tau_1), \ldots, \text{apply-intf}(\mathbf{v}_n, \tau_n)], \ell' \rangle & \tau = \texttt{[}\tau_1\texttt{,} \ldots \texttt{,}\tau_n\texttt{]} \wedge v = [\mathbf{v}_1, \ldots, \mathbf{v}_n] \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $\ell' \triangleq \ell \sqcup \langle \text{public}, \text{tainted} \rangle$

apply-intf$(\mathbf{v}, \tau)$ checks that $\mathbf{v}$ has type τ and, if $\mathbf{v}$ is a tuple, propagates the interface to the components of $\mathbf{v}$.

Going on, we have rules for unary and binary operations (we assume to have a function $[\![\cdot]\!] : \text{Bop} \to (\text{Val} \to \text{Val} \to \text{Val})$ that gives the semantic interpretation of each binary operation, and equivalently one for unary operations) and if expressions:

$$\frac{\Delta, pc \vdash e \Downarrow \langle v,\ \ell \rangle}{\Delta, pc \vdash uop\ e \Downarrow \langle [\![uop]\!](v),\ pc \sqcup \ell \rangle}\ [\text{Uop}]$$

$$\frac{\Delta, pc \vdash e_1 \Downarrow \langle v_1,\ \ell_1 \rangle \quad \Delta, pc \vdash e_2 \Downarrow \langle v_2,\ \ell_2 \rangle}{\Delta, pc \vdash e_1\ bop\ e_2 \Downarrow \langle [\![bop]\!](v_1, v_2),\ pc \sqcup \ell_1 \sqcup \ell_2 \rangle}\ [\text{Bop}]$$

$$\frac{\Delta, pc \vdash e_c \Downarrow \langle \mathbf{tt},\ \ell_c \rangle \quad \Delta, pc \sqcup \ell_c \vdash e_t \Downarrow \langle v,\ \ell_t \rangle}{\Delta, pc \vdash \texttt{if}\ e_c\ \texttt{then}\ e_t\ \texttt{else}\ e_e \Downarrow \langle v,\ pc \sqcup \ell_c \sqcup \ell_t \rangle}\ [\text{IfTrue}]$$

$$\frac{\Delta, pc \vdash e_c \Downarrow \langle \mathbf{ff},\ \ell_c \rangle \quad \Delta, pc \sqcup \ell_c \vdash e_e \Downarrow \langle v,\ \ell_e \rangle}{\Delta, pc \vdash \texttt{if}\ e_c\ \texttt{then}\ e_t\ \texttt{else}\ e_e \Downarrow \langle v,\ pc \sqcup \ell_c \sqcup \ell_e \rangle}\ [\text{IfFalse}]$$

Let bindings also perform attribute checking and casting. `let` *attrs* x `=` e_1 `in` e_2 evaluates e_1, checks that its security label is compatible with attributes *attrs*, then

upcasts it to *attrs* if it was more permissive, and binds the result to x in the evaluation of e_2:

$$\frac{\Delta, pc \vdash e_1 \Downarrow \langle v_1,\ \ell_1\rangle \quad p \triangleq \Delta(\texttt{@plugin}) \quad \ell_1 \sqsubseteq [\![0; a_1, \ldots, a_n]\!]^{\top}_{\text{attr}} \quad \Delta[x \mapsto \langle v_1, \ell_1 \sqcup [\![p; a_1, \ldots, a_n]\!]^{\bot}_{\text{attr}}\rangle], pc \vdash e_2 \Downarrow \langle v,\ \ell\rangle}{\Delta, pc \vdash \texttt{let}\ a_1\ \ldots\ a_n\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow \langle v,\ pc \sqcup \ell\rangle}\ [\text{Let}]$$

The functions $[\![\cdot]\!]^{\top}_{\text{attr}}$ and $[\![\cdot]\!]^{\bot}_{\text{attr}}$ are defined as:

$$\begin{aligned}
[\![p; a_1, \ldots, a_n]\!]^{\top}_{\text{attr}} &= \bigsqcap_{i=1} [\![p; a_i]\!]^{\top}_{\text{attr}} && n \neq 1 \\
[\![p; a_1, \ldots, a_n]\!]^{\bot}_{\text{attr}} &= \bigsqcup_{i=1} [\![p; a_i]\!]^{\bot}_{\text{attr}} && n \neq 1 \\
[\![p; \texttt{secret}]\!]^{X}_{\text{attr}} &= (\text{secret}^{p}, X_I) \\
[\![p; \texttt{public}]\!]^{X}_{\text{attr}} &= (\text{public}, X_I) \\
[\![p; \texttt{tainted}]\!]^{X}_{\text{attr}} &= (X_C, \text{tainted}) \\
[\![p; \texttt{untainted}]\!]^{X}_{\text{attr}} &= (X_C, \text{untainted})
\end{aligned}$$

Note the meet and join of 0 elements are $\top$ and $\bot$, respectively.

Essentially, $[\![p; \textit{attrs}]\!]^{X}_{\text{attr}}$ turns a list of attributes into an element of Lbl; if $X = \top$, the result is the smallest element of the lattice compatible with *attrs*, if $X = \bot$ it is the greatest. We use $X = \top$ when checking if the assignment is valid so that `let public secret x = <secret value>` is rejected, and $X = \bot$ when casting so that with `let public secret x = <public value>`, `x` is secret in the body.

With a slight change of the semantics, i.e. always using $X = \top$, we could implement a *strict mode* for the language: in that case, all `let`s with unspecified attributes become equivalent to `let secret tainted`. This is a sort of *secure-by-default* stance and forces the programmer to explicitly state when a variable needs to be public or when it is untainted. This choice is similar to that of having immutable values by default and having to explicitly mark mutable ones, and the argument against it is that it makes variable definition more verbose.

There is also a rule that skips the evaluation of a deferred expression if it is immediatly bound in a `let`; this is needed to avoid divergence in the implementation of `let rec`:

$$\frac{\begin{array}{c}\Delta, pc \vdash e_t \Downarrow \langle [\langle v_j, \ell_{v_j}\rangle]_{j\in[0,\ldots,n-1]},\ \ell_t\rangle \quad \Delta, pc \vdash e_i \Downarrow \langle i,\ \ell_i\rangle \\ 0 \leq i \leq n-1 \quad \ell'_i \triangleq \ell_{v_i} \sqcup \ell_i \\ p \triangleq \Delta(\texttt{@plugin}) \quad \ell'_i \sqsubseteq [\![0; a_1, \ldots, a_n]\!]^{\top}_{\text{attr}} \\ \Delta[x \mapsto \langle v_i, \ell'_i \sqcup [\![p; a_1, \ldots, a_n]\!]^{\bot}_{\text{attr}}\rangle], pc \vdash e_2 \Downarrow \langle v,\ \ell\rangle\end{array}}{\Delta, pc \vdash \texttt{let}\ a_1\ \ldots\ a_n\ x\ \texttt{=}\ e_t\texttt{[}e_i\texttt{]}\ \texttt{in}\ e_2 \Downarrow \langle v,\ pc \sqcup \ell\rangle}\ [\text{LetProj}]$$

We also have fixpoint operators `fix` and `fix*` (for mutually recursive definitions),

which are also used to implement `let rec ... and` definitions:

$$\frac{\Delta, pc \vdash e \Downarrow \langle \lambda^{\Delta'} x.\ e',\ \ell \rangle \quad \Delta'[x \mapsto \langle \mathrm{defer}^{\Delta'}\ \texttt{fix}\ \lambda x.\ e', \ell \rangle], pc \sqcup \ell \vdash e' \Downarrow \langle v,\ \ell' \rangle}{\Delta, pc \vdash \texttt{fix}\ e \Downarrow \langle v,\ pc \sqcup \ell \sqcup \ell' \rangle}\ [\mathrm{Fix}]$$

$$\frac{\begin{array}{c} \Delta, pc \vdash e \Downarrow \langle [\overbrace{\langle \lambda^{\Delta'_i} x_i.\ e'_i,\ \ell_i \rangle}^{f_i}]_{i \in [1,n]},\ \ell \rangle \\ \Delta'' \triangleq \Delta[y_1 \mapsto f_1, \ldots, y_n \mapsto f_n] \quad y_i \text{ fresh} \\ \forall i \in [1,n].\ \Delta'_i[x_i \mapsto \langle [\langle \mathrm{defer}^{\Delta''}(\texttt{fix*}\ \texttt{[}y_1\texttt{,}\ldots\texttt{,}y_n\texttt{]})\texttt{[}j\texttt{]},\ \ell_j \rangle]_{j \in [1,n]},\ \ell_i \rangle], pc \sqcup \ell_i \vdash e'_i \Downarrow \overbrace{\langle v_i,\ \ell'_i \rangle}^{\mathbf{v}_i} \end{array}}{\Delta, pc \vdash \texttt{fix*}\ e \Downarrow \langle [\mathbf{v}_1, \ldots, \mathbf{v}_n],\ pc \sqcup \ell \sqcup \bigsqcup_{i=1}^{n} \ell_i \sqcup \bigsqcup_{i=1}^{n} \ell'_i \rangle}\ [\mathrm{Fix*}]$$

$$\frac{\begin{array}{c} xs \text{ fresh} \quad \mathrm{wrap}(e') \triangleq \lambda xs.\ (\texttt{let}\ a^i_1 \ldots a^i_{n_i}\ x_i\ \texttt{=}\ xs\texttt{[}i-1\texttt{]}\ \texttt{in})_{i \in [1,m]}\ e' \\ \Delta, pc \vdash \texttt{fix*}\ \texttt{[}\mathrm{wrap}(e_1)\texttt{,}\ldots\texttt{,}\mathrm{wrap}(e_m)\texttt{]} \Downarrow \langle [\langle v_i, \ell_i \rangle]_{i \in [1,m]},\ \ell \rangle \\ p \triangleq \Delta(\text{@plugin}) \quad \forall i \in [1,m].\ \ell_i \sqsubseteq [\![p; a^i_1, \ldots, a^i_{n_i}]\!]^{\top}_{\mathrm{attr}} \\ \mathbf{u}_i \triangleq \langle v_i, \ell_i \sqcup [\![p; a^i_1, \ldots, a^i_{n_i}]\!]^{\bot}_{\mathrm{attr}} \rangle \quad \Delta[x_1 \mapsto \mathbf{u}_1, \ldots, x_m \mapsto \mathbf{u}_m], pc \vdash e \Downarrow \langle v,\ \ell' \rangle \end{array}}{\Delta, pc \vdash \texttt{let rec}\ a^1_1\ \ldots\ a^1_{n_1}\ x_1 = e_1\ (\texttt{and}\ a^i_1\ \ldots\ a^i_{n_i}\ x_i = e_i)_{i \in [2,m]}\ \texttt{in}\ e \Downarrow \langle v,\ pc \sqcup \ell' \rangle}\ [\mathrm{LetRecAnd}]$$

Modules are first class values; at runtime they are environments that can be passed around and accessed dynamically. To evaluate a module, we translate it to a chain of `let` expressions. The body of the final `let` is the expression $\lambda x.\ x$; this is a trick to be able to extract the environment at the end of the evaluation of the module (since the lambda will produce a closure).

Only the identifiers which are explicitly exported are accessible outside the module. This is effectively achieved by restricting the environment resulting from the evaluation of the environment expression.

We keep track of whether the module was a simple `module` or a `trusted module` through a binding of the special identifier @trusted.

$$\frac{\Delta[\text{@trusted} \mapsto \mathbf{ff}], pc \vdash [\![d_1, \ldots, d_n]\!]_{\mathrm{mod} \to \mathrm{let}} \Downarrow \langle \lambda^{\Delta'} x.\ x,\ \ell \rangle}{\Delta, pc \vdash \texttt{module}\ d_1 \ldots d_n\ \texttt{end} \Downarrow \langle \mathrm{mod}\ \Delta'_{|\{x\ |\ \texttt{export}\ x \in \{d_1,\ldots,d_n\}\}},\ pc \rangle}\ [\mathrm{Module}]$$

$$\frac{\Delta[\text{@trusted} \mapsto \mathbf{tt}], pc \vdash [\![d_1, \ldots, d_n]\!]_{\mathrm{mod} \to \mathrm{let}} \Downarrow \langle \lambda^{\Delta'} x.\ x,\ \ell \rangle}{\Delta, pc \vdash \texttt{trusted module}\ d_1 \ldots d_n\ \texttt{end} \Downarrow \langle \mathrm{tmod}\ \Delta'_{|\{x\ |\ \texttt{export}\ x \in \{d_1,\ldots,d_n\}\}},\ pc \rangle}\ [\mathrm{TrustedModule}]$$

This is the definition of the translation from module syntax to let expressions:

$$[\![\varepsilon]\!]_{\mathrm{mod} \to \mathrm{let}} \triangleq \lambda x.\ x$$

$$[\![\texttt{export}\ x, d_1, \ldots, d_n]\!]_{\mathrm{mod} \to \mathrm{let}} \triangleq [\![d_1, \ldots, d_n]\!]_{\mathrm{mod} \to \mathrm{let}}$$

$$[\![\texttt{let}\ attr*\ x = e, d_1, \ldots, d_n]\!]_{\mathrm{mod} \to \mathrm{let}} \triangleq \texttt{let}\ attr*\ x\ \texttt{=}\ e\ \texttt{in}\ [\![d_1, \ldots, d_n]\!]_{\mathrm{mod} \to \mathrm{let}}$$

$$\left[\!\!\left[\begin{array}{l} \texttt{let rec}\ a^1_1\ \ldots\ a^1_{n_1}\ x_i = e_1 \\ (\texttt{and}\ a^i_1\ \ldots\ a^i_{n_i}\ x_i = e_i)_{i \in [2,m]}, \\ d_1, \ldots, d_n \end{array} \right\} l \right]\!\!\right]_{\mathrm{mod} \to \mathrm{let}} \triangleq l\ \texttt{in}\ [\![d_1, \ldots, d_n]\!]_{\mathrm{mod} \to \mathrm{let}}$$

`plugin` expressions are composed of a file name and a list of interfaces. The file is parsed and evaluated, resulting in a module (its content is implicitly wrapped with `module` ... `end`). Then, we apply the specified interfaces to the bindings contained in that module, discarding the bindings that do not have one.

$$\frac{\begin{array}{c} p \text{ fresh plugin number} \\ \varnothing[\text{@plugin} \mapsto p], \bot \vdash \text{read-file}(f) \Downarrow \langle \text{mod } \Delta',\ \ell \rangle \\ \Delta''(x) \triangleq \begin{cases} \text{apply-intf}(\Delta'(x), \tau) & (x : \tau) \in \{i_1, \ldots, i_n\} \\ \text{undefined} & \text{otherwise} \end{cases} \end{array}}{\Delta, pc \vdash \texttt{plugin}\ f\ i_1 \ldots i_n\ \texttt{end} \Downarrow \langle \text{plug } \Delta'',\ pc \sqcup \ell \rangle}\ [\text{Plugin}]$$

Finally, names defined in modules and plugins can be accessed through their environment:

$$\frac{\Delta, pc \vdash e \Downarrow \langle m\ \Delta',\ \ell \rangle \quad \Delta'(x) = \langle v, \ell' \rangle \quad m \in \{\text{mod}, \text{tmod}, \text{plug}\}}{\Delta, pc \vdash e.x \Downarrow \langle v,\ pc \sqcup \ell \sqcup \ell' \rangle}\ [\text{FieldAccess}]$$

We provide operators for declassifying and endorsing values as well as the control flow label (pc). Those operators can only be used inside trusted modules. Furthermore declassification of a secret^p value can only be performed inside the plugin p or the main module 0.

$$\frac{\Delta, pc \vdash e \Downarrow \langle v,\ \langle \ell_C, \ell_I \rangle \rangle \quad \Delta(\text{@trusted}) = \mathbf{tt} \quad \ell_C = \text{secret}^p \Rightarrow \Delta(\text{@plugin}) \in \{0, p\}}{\Delta, pc \vdash \texttt{declassify}\ e \Downarrow \langle v,\ \langle \text{public},\ \ell_I \rangle \rangle}\ [\text{Declassify}]$$

$$\frac{\Delta, pc \vdash e \Downarrow \langle v,\ \langle \ell_C, \ell_I \rangle \rangle \quad \Delta(\text{@trusted}) = \mathbf{tt}}{\Delta, pc \vdash \texttt{endorse}\ e \Downarrow \langle v,\ \langle \ell_C,\ \text{untainted} \rangle \rangle}\ [\text{Endorse}]$$

$$\frac{pc_C = \text{secret}^p \Rightarrow \Delta(\text{@plugin}) \in \{0, p\} \quad \Delta(\text{@trusted}) = \mathbf{tt} \quad \Delta, \langle \text{public}, pc_I \rangle \vdash e \Downarrow \langle v,\ \langle \ell_C,\ \ell_I \rangle \rangle}{\Delta, \langle pc_C, pc_I \rangle \vdash \texttt{declassify\_pc}\ e \Downarrow \langle v,\ \langle \ell_C,\ pc_I \sqcup \ell_I \rangle \rangle}\ [\text{DeclassifyPC}]$$

$$\frac{\Delta(\text{@trusted}) = \mathbf{tt} \quad \Delta, \langle pc_C, \text{untainted} \rangle \vdash e \Downarrow \langle v,\ \langle \ell_C,\ \ell_I \rangle \rangle}{\Delta, \langle pc_C, pc_I \rangle \vdash \texttt{endorse\_pc}\ e \Downarrow \langle v,\ \langle pc_C \sqcup \ell_C,\ \ell_I \rangle \rangle}\ [\text{EndorsePC}]$$

`has_attr` can be used to check whether a value has a certain attribute. `assert` e halts the program if e evaluates to false. `die` has no rules, thus it always halts the

program.

$$\frac{\begin{array}{c}\Delta, pc \vdash e \Downarrow \langle v,\ \langle \ell_C, \ell_I \rangle \rangle \\ b \triangleq \quad (attr = \texttt{public} \wedge \ell_C = \text{public}) \vee (attr = \texttt{secret} \wedge \exists p.\ \ell_C = \text{secret}^p) \\ \vee\ (attr = \texttt{untainted} \wedge \ell_I = \text{untainted}) \vee (attr = \texttt{tainted} \wedge \ell_I = \text{tainted})\end{array}}{\Delta, pc \vdash \texttt{has\_attr}\ attr\ e \Downarrow \langle b,\ pc \sqcup \langle \ell_C, \ell_I \rangle \rangle}\ [\text{HasAttr}]$$

$$\frac{\Delta, pc \vdash e \Downarrow \langle \mathbf{tt},\ \ell \rangle}{\Delta, pc \vdash \texttt{assert}\ e \Downarrow \langle \mathbf{tt},\ pc \sqcup \ell \rangle}\ [\text{Assert}]$$

To avoid information leak, it would be ideal to prevent usage of `assert` and `die` in secret contexts, since program termination is a covert information channel. However, we chose not to since they can always be implemented as functions:

```
let my_assert guard =
    if guard then
        true
    else
        # crash exploiting a security violation
        let public x =
            (let secret x = 1 in x)
        in []
```

and we have no way to prevent program termination in secret contexts in all cases.

Finally, we have operators for simple input and output:

$$\frac{\Delta, pc \vdash e \Downarrow \langle v,\ \ell \rangle \quad pc \sqcup \ell = \langle \text{public}, \ell_I \rangle}{\Delta, pc \vdash \texttt{print}\ e \Downarrow \langle v,\ pc \sqcup \ell \rangle}\ [\text{Print}]$$

$$\frac{v \triangleq \text{user input}}{\Delta, pc \vdash \texttt{get} \Downarrow \langle v,\ pc \sqcup \langle \text{public}, \text{tainted} \rangle \rangle}\ [\text{Get}]$$

The initial environment contains the bindings $\text{plugin} \mapsto 0, \text{trusted} \mapsto \mathbf{ff}$.

3 Effect handlers

We initially decided to implement effect handlers for the language[4] and redefine security exceptions as operations. This would make for a more flexible language, where the program does not necessarily crash when a security violation happens. For example, one could implement a password manager that returns the password in secret contexts, but when an exception is raised on accessing the password from

[4]Realistically, could we make a language and not put algebraic effects in it?

a public context, the manager would return an obfuscated version of the password and resume the execution. An effect system would also make for some interesting considerations regarding control and information flow, for example on whether we need to consider the invocation itself of operations a covert channel, as it might be dependent on some data inputted by an attacker.

Unfortunately, it was not possible to implement handlers during this timeframe. Nevertheless, here are some prototype rules for basic handler definition, operation raising and handling, and to build continuations for simple constructs such as binary operations.

$$\frac{\forall i \in [1, n].\ \Delta, pc \vdash e_i \Downarrow \mathbf{v}_i \quad y \text{ fresh}}{\Delta, pc \vdash \texttt{do}\ op\ e_1\ \ldots\ e_n \Downarrow \langle op, \lambda^{\Delta} y.\ y, \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle, pc \rangle} \text{[Do]}$$

$$\frac{\Delta, pc \vdash e \Downarrow \mathbf{v} \quad (\texttt{return}\ x = e_{\text{ret}}) \in h \quad \Delta[x \mapsto \mathbf{v}], pc \vdash e_{\text{ret}} \Downarrow \nu}{\Delta, pc \vdash \texttt{with}\ h\ \texttt{handle}\ e \Downarrow \nu} \text{[HandleRet]}$$

$$\frac{\Delta, pc \vdash e \Downarrow \langle op, c, \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle, pc' \rangle \quad (op\ x_1\ \ldots\ x_n\ k = e_{\text{op}}) \notin h \quad y \text{ fresh}}{\Delta, pc \vdash \texttt{with}\ h\ \texttt{handle}\ e \Downarrow \langle op, \lambda^{\Delta} y.\ \texttt{with}\ h\ \texttt{handle}\ c\ y, \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle, pc' \rangle} \text{[HandleRaise]}$$

$$\frac{\begin{array}{c}\Delta, pc \vdash e \Downarrow \langle op, c, \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle, pc' \rangle \\ (op\ x_1\ \ldots\ x_n\ k = e_{\text{op}}) \in h \\ c' = \lambda^{\Delta} y.\ \texttt{with}\ h\ \texttt{handle}\ c\ y \quad y \text{ fresh} \\ \Delta[x_1 \mapsto \mathbf{v}_1, \ldots, x_n \mapsto \mathbf{v}_n, k \mapsto \langle c', pc' \rangle], pc' \vdash e_{\text{op}} \Downarrow \nu\end{array}}{\Delta, pc \vdash \texttt{with}\ h\ \texttt{handle}\ e \Downarrow \nu} \text{[HandleOp]}$$

$$\frac{\Delta, pc \vdash e_1 \Downarrow \langle op, c, \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle, pc' \rangle \quad y \text{ fresh}}{\Delta, pc \vdash e_1\ bop\ e_2 \Downarrow \langle op, \lambda^{\Delta} y.\ (c\ y)\ bop\ e_2, \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle, pc' \rangle} \text{[BopRaiseL]}$$

$$\frac{\Delta, pc \vdash e_1 \Downarrow \mathbf{v} \quad \Delta, pc \vdash e_2 \Downarrow \langle op, c, \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle, pc' \rangle \quad x, y \text{ fresh}}{\Delta, pc \vdash e_1\ bop\ e_2 \Downarrow \langle op, \lambda^{\Delta[x \mapsto \mathbf{v}]} y.\ x\ bop\ (c\ y), \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle, pc' \rangle} \text{[BopRaiseR]}$$

4 Executing the interpreter

These instructions assume that `opam` is installed and that `make` is available on the machine.

1. Clone the git repository at `git@github.com:Alex23087/Scaml2.git`
2. Go into the `src` folder, and run `make setup` to install the required OCaml packages
3. Run `make test-interpreter` to run the unit tests

4. Run `make exec file=`***`filename`*** to execute a ScaML^2 program saved in *filename*.

5. In the `example` folder, run the script `run.sh` to run an example program.