Notes on: A Logic of Programs with Interface-confined Code

Forrest Li, Spring 2017

Introduction

Interface-confinement is a sandbox mechanism by which system resources are restricted behind a set of defined interfaces. When running untrusted code in the sandbox, the code can only request access to those resources that are behind the available interfaces.

System M is a program logic for modeling, proving and reasoning about safety properties of a system under analysis. The following sections outline the structure of System M.

System M

Term syntax:

$$
\begin{array}{llcl}
\textit{Base values} & bv & ::= & \texttt{tt} \mid \texttt{ff} \mid \iota \mid \ell \mid n \mid (\,) \\
\textit{Expressions} & e & ::= & x \mid bv \mid \lambda x.e \mid \texttt{fix}\ f(x).e \\
 & & \mid & \Lambda X.e \mid e_1\ e_2 \mid e\ \cdot \mid \texttt{comp}(c) \\
\textit{Actions} & a & ::= & A \mid a\ e \mid a\ \cdot \\
\textit{Computations} & c & ::= & \texttt{act}(a) \mid \texttt{ret}(e) \\
 & & \mid & \texttt{letc}(c_1, x.c_2) \mid \texttt{lete}(e_1, x.c_2) \\
 & & \mid & \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2
\end{array}
$$

- *Base values*: true, false, thread ID, memory location, integer
- *Expressions*: variables, base values, varying types of functions, func application, suspended computation
- *Actions*: read, write, check
- *Computations*: atomic action, return, sequential composition, branching

Concurrent system configuration syntax:

$$
\begin{array}{llcl}
\textit{Stack} & K & ::= & [] \mid x.c :: K \\
\textit{Thread} & T & ::= & \langle \iota; K; c\rangle \mid \langle \iota; K; e\rangle \mid \langle \iota; \mathsf{stuck}\rangle \\
\textit{Configuration} & C & ::= & \sigma \triangleright T_1, \ldots, T_n
\end{array}
$$

- *Stack*: a stack of frames. A frame is *x.c*, where x binds the return expression of the computation preceding c.
- *Thread*: A triple <Thread_ID, Stack, Computation or Expression or Stuck>
    - Stuck means thread performed illegal action
- *Configuration*: shared state σ that presides over all threads in a system

System M operates on **small-step transitions**, which advance the system by one computation step. Small-step transitions are defined by the relation $\sigma \triangleright T \hookrightarrow \sigma' \triangleright T'$. A sample of the transitions are provided below.

$$\boxed{\sigma \triangleright T \hookrightarrow \sigma' \triangleright T'}$$

$$\frac{\mathsf{next}(\sigma, a) = (\sigma', e) \qquad e \neq \mathsf{stuck}}{\sigma \triangleright \langle \iota; x.c :: K; \mathtt{act}(a) \rangle \hookrightarrow \sigma' \triangleright \langle \iota; K; c[e/x] \rangle} \text{ R-ACTS}$$

$$\frac{\mathsf{next}(\sigma, a) = (\sigma', \mathsf{stuck})}{\sigma \triangleright \langle \iota; x.c :: K; \mathtt{act}(a) \rangle \hookrightarrow \sigma' \triangleright \langle \iota; \mathsf{stuck} \rangle} \text{ R-ACTF}$$

$$\frac{}{\sigma \triangleright \langle \iota; \mathsf{stuck} \rangle \hookrightarrow \sigma \triangleright \langle \iota; \mathsf{stuck} \rangle} \text{ R-STUCK}$$

$$\frac{}{\sigma \triangleright \langle \iota; x.c :: K; \mathtt{ret}(e) \rangle \hookrightarrow \sigma \triangleright \langle \iota; K; c[e/x] \rangle} \text{ R-RET}$$

$$\frac{}{\sigma \triangleright \langle \iota; K; \mathtt{lete}(e_1, x.c_2) \rangle \hookrightarrow \sigma \triangleright \langle \iota; x.c_2 :: K; e_1 \rangle} \text{ R-SEQE1}$$

$$\frac{e \rightarrow_\beta e'}{\sigma \triangleright \langle \iota; K; e \rangle \hookrightarrow \sigma \triangleright \langle \iota; K; e' \rangle} \text{ R-SEQE2}$$

$$\frac{}{\sigma \triangleright \langle \iota; K; \mathtt{comp}(c_1) \rangle \hookrightarrow \sigma \triangleright \langle \iota; K; c_1 \rangle} \text{ R-SEQE3}$$

- *R-ACTS*: next() returns a new shared state and the resulting expression from *a*. If *e* is not the stuck expression, then the new thread state pops off the frame *x.c*, *e* binds to *x*, and *c* is the next state.
  - This is like saying *c(x=a())*, where *x* is *c*'s argument.
    - The process of binding *e* to *x* and then evaluating the expression is *beta reduction*.
- *R-ACTF*: if next() returns a stuck expression, then the thread becomes stuck.

A **trace** is is a finite sequence of reductions, which aligns with the notion of thread execution.

A **time point** is an natural number (i.e. clock time) associated with a reduction in a trace. Time points are monotonically increasing over a trace.

Interface Confinement Example: Counter

A simple interface to increment, get, and execute-and-print a counter value in memory. The desired *invariant is cnt can never decrease.*

$$\begin{aligned}
inc &= \mathtt{comp}(\mathtt{letc}\ x = \mathsf{read}\ cnt; \mathsf{write}\ cnt\ (x{+}1)) \\
get &= \mathtt{comp}(\mathtt{letc}\ x = \mathsf{read}\ cnt; \mathtt{ret}(x)) \\
prn &= \lambda y.\mathtt{comp}(\mathtt{lete}\ _ = y \\
&\qquad\qquad \mathtt{letc}\ x = \mathsf{read}\ \ cnt;\ \mathsf{print}\ \ x) \\
c &= \mathtt{letc}\ x = \mathsf{download}(); \mathtt{letc}\ y = \mathsf{check}\ x; \\
&\quad\ \mathtt{lete}\ _ = y\ inc\ get\ prn; \mathtt{ret}()
\end{aligned}$$

- *inc*: increment counter
- *get*: get counter value
- *prn*: execute code *y*, then print counter.
- *c*: a computation involving downloading untrusted code and running it with the *inc*,*get*,*prn* interfaces.
  - “check” makes sure the code doesn’t have any actions that can modify *cnt* directly.
  - Given the three interfaces, *cnt* should never decrease.

Types

System M defines a set of **types**.

$$
\begin{array}{llll}
\textit{Base types} & \mathtt{b} & ::= & \mathtt{bool} \mid \mathtt{nat} \mid \mathtt{unit} \mid \mathtt{ptr} \mid \mathtt{time} \mid \mathtt{thread} \\
\textit{Expr types} & \tau & ::= & X \mid \mathtt{b} \mid \Pi x{:}\tau_1.\tau_2 \mid \forall X.\tau \mid \mathtt{comp}(\eta_c) \\
 & & \mid & \mathtt{any} \mid \mathtt{inv}(\Xi.\varphi) \mid \mathtt{FAE} \\
\textit{Comp types} & \eta & ::= & (x{:}\tau.\varphi, \varphi') \\
\textit{Closed c types} & \eta_c & ::= & (\Xi.x{:}\tau.\varphi_1, \Xi.\varphi_2) \\
\textit{Assertions} & \varphi & ::= & P \mid e_1 = e_2 \mid \varphi\ e \mid \top \mid \bot \mid \neg\varphi \\
 & & \mid & \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \forall x{:}\tau.\varphi \mid \exists x{:}\tau.\varphi \\
\textit{Expressions} & e & ::= & \cdots \mid \mathsf{self} \\
\textit{Exec ctx} & \Xi & ::= & (u_b : \mathtt{time}, u_e : \mathtt{time})
\end{array}
$$

- The type of a variable *x* is denoted as *x:type*.
- Expr type: type variables, base types, dependent function types, invariant types
  - FAE describes expressions that syntactically don’t have any action symbols.
- Computation type: a pair (<partial correctness assertion, invariant assertion>)
  - Partial correctness asserts a computation, if it terminates, will satisfy φ_1 and have return type τ .
  - Invariant asserts the effects of computation during evaluation.
- The “self” expression refers to the current thread evaluating an action or expression.
- Execution context: a tuple of (<start time>, <end time>)

An important expression type is **inv(Ξ.φ)**. An expression of this type, when evaluated, preserves invariant **φ** over the execution context **Ξ**.

Using the Counter example, here is an invariant that says the counter *cnt* never decreases in the time interval $[u_b, u_e]$.

$$
\begin{aligned}
\varphi_{nd}(u_b, u_e) = \forall t_1, t_2, l, v_1, v_2, u_b \le t_1 < t_2 \le u_e \wedge \mathsf{eval}\ cnt\ l \\
\mathsf{mem}\ l\ v_1\ t_1 \wedge \mathsf{mem}\ l\ v_2\ t_2 \Rightarrow v_2 \ge v_1
\end{aligned}
$$

- *eval cnt l*: a predicate that is true if expr *cnt* beta-reduces to *expr l*. That is, *cnt* is a memory location expression that reduces to *l*.
- *mem l v t*: a predicate that is true if memory location *l* has value *v* at time *t*.

Type Semantics

The *interpretation* of an expression type \tau is a **semantic type,** C.
C is a set of pairs, (<step index>, <expression>).

- Step index is a number associated with a reduction step.
- Expression must be in *normal form*, meaning it cannot be reduced any further.

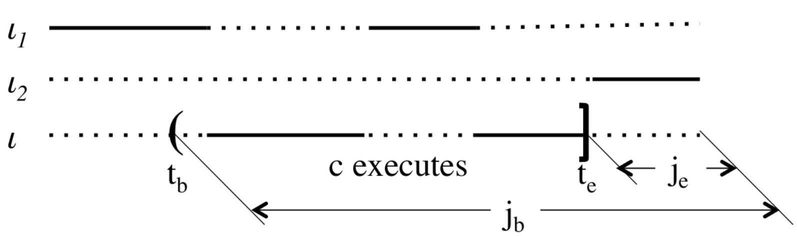
The set of all semantic types, `Type`:

$$\mathsf{Type} \stackrel{def}{=} \{\mathsf{C} \mid \mathsf{C} \in \mathcal{P}(\{(j, \mathtt{nf}) \mid j \in \mathbb{N}\}) \wedge (\forall k, \mathtt{nf}, (k, \mathtt{nf}) \in \mathsf{C} \wedge j < k \Longrightarrow (j, \mathtt{nf}) \in \mathsf{C})\}$$

Interpretation of Computation Types

Interpretation of a computation type is denoted $\mathcal{RC}[\![\eta]\!]^{\mathcal{K},\iota}_{\theta;\mathcal{T}}$.

- This is a set of step-indexed computations, which are pairs *(k, c)*.
- θ is a partial map from type variables to `Type`.
- $K=(t_b, t_e)$, is the time interval in which *c* executes.
- ι is thread identifier
- *T* is the trace
- The following graphic shows an example trace *T* and the symbols in $\mathcal{RC}[\![\eta]\!]^{\mathcal{K},\iota}_{\theta;\mathcal{T}}$.



Interpretation of Expression Types

Two interpretations:

- *value interpretation*, $\mathcal{RV}[\![\tau]\!]$
  - *RV[any]*: set of all pairs of *(k, nf)*, where *k* is a natural number.
  - *RV[X], X* is type variable: θ(X)
  - *RV[inv(Ξ.φ)]*: union of (1) stuck terms, (2) suspended computations, (3) indexed functions, (4) recursive functions, and (5) polymorphic functions.
- *expression interpretation*, $\mathcal{RE}[\![\tau]\!]$
  - $\mathcal{RE}[\![\tau]\!]$ lifts the value interpretation $\mathcal{RV}[\![\tau]\!]$. Each pair *(k, e)* satisfies that, for *j*≤*k*, *e* reduces to beta normal form *e'* in *j* steps, and that *(k-j, e')* is in $\mathcal{RV}[\![\tau]\!]$.

Formula Semantics

Formulas are interpreted over traces *T*.

A sample of formula semantics is listed below.

- *ε(T)* is the set of atomic formulas that are true over T.

- *start(e_1, comp(c), e_2)* is a predicate that execution of computation c by the thread whose ID e_1 reduces to starts at e_2.

$$\mathcal{T} \vDash P\ \vec{e} \text{ iff } P\ \vec{e} \in \varepsilon(\mathcal{T})$$
$$\mathcal{T} \vDash \mathsf{start}(e_1, \mathtt{comp}(c), e_2) \text{ iff } e_1 \to_\beta^* \iota \not\to_\beta, e_2 \to_\beta^* t \not\to_\beta,$$
$$\text{and thread } \iota \text{ has } c \text{ as the active computation with}$$
$$\text{an empty stack at time } t \text{ on } \mathcal{T}$$
$$\mathcal{T} \vDash \forall x{:}\tau.\varphi \text{ iff } \forall e, (\_, e) \in [\![\tau]\!] \text{ implies } \mathcal{T} \vDash \varphi[e/x]$$
$$\mathcal{T} \vDash \exists x{:}\tau.\varphi \text{ iff } \exists e, (\_, e) \in [\![\tau]\!] \text{ and } \mathcal{T} \vDash \varphi[e/x]$$

Type System and Assertion Logic

Several environment contexts for typing judgements:

- Θ: type variables
- Σ: specifications for action symbols
- Γ: type bindings
- Δ: logical assertions
- Ξ: compuation execution time interval
- `self`: the ID of the thread currently executing a computation

Examples of typing judgements are listed below.

$u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \tau$	expression e has type τ
$\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash c : \eta$	computation c has type η
$\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash \varphi\ \mathsf{silent}$	φ holds while reductions are non-effectful
$\Theta; \Sigma; \Gamma; \Delta \vdash \varphi\ \mathsf{true}$	φ is true

Silent threads

- Threads that either perform non-effectful reductions or do not perform reductions at all.
- The following states that if the invariant is true and the invariant is closed under (Ξ,Γ), then the invariant holds under non-effectful reductions/no reductions.

$$\frac{\Theta; \Sigma; \Xi, \Gamma; \Delta \vdash \varphi\ \mathsf{true} \qquad \Xi, \Gamma \vdash \varphi\ \mathsf{ok}}{\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash \varphi\ \mathsf{silent}}\ \textsc{Silent}$$

Computation Typing

Typing of computations, with the possible computations below.

- *act(a)*
- *ret(e)*
- *letc(c1, x.c2)*
- *lete(e, x.c)*
- *if e then c_1 else c_2*

A sampling of the ACT and RET typing rules are listed below.

$$\frac{\begin{array}{c}\Xi;\Theta;\Sigma;\Gamma;\Delta \vdash a :: \mathtt{Act}(\Xi'.x{:}\tau.\varphi_1, \Xi'.\varphi_2) \\ \Xi;\Theta;\Sigma;\Gamma;\Delta \vdash \varphi \ \textsf{silent} \qquad \mathtt{fv}(a) \in \mathtt{dom}(\Gamma)\end{array}}{\Xi;\Theta;\Sigma;\Gamma;\Delta \vdash \mathtt{act}(a) : (x{:}\tau.\varphi_1[\Xi/\Xi'], \varphi_2[\Xi/\Xi'] \wedge \varphi)} \ \textsc{Act}$$

$$\frac{\begin{array}{c}E(\Xi);\Theta;\Sigma;B(\Xi);\Gamma;\Delta \vdash e : \tau \\ \Xi;\Theta;\Sigma;\Gamma;\Delta \vdash \varphi \ \textsf{silent} \qquad \mathtt{fv}(e) \subseteq \mathtt{dom}(\Gamma)\end{array}}{\Xi;\Theta;\Sigma;\Gamma;\Delta \vdash \mathtt{ret}(e) : (x{:}\tau.((x = e) \wedge \varphi), \varphi)} \ \textsc{Ret}$$

- Act(…) draws actions from specifications from Σ.
  - Sample actions: read, write, and check.
  - Example: the type for the read action takes a memory location as argument.

Expression Typing

Typing of expressions involve assigning types to expressions. Examples below show typing to ANY and typing to INV.

$$\frac{\Theta;\Sigma;u,\Gamma \vdash \Delta \ \textsf{ok} \qquad \mathtt{fv}(e) \subseteq \mathtt{dom}(\Gamma)}{u;\Theta;\Sigma;\Gamma;\Delta \vdash e : \mathtt{any}} \ \textsc{Any}$$

$$\frac{\begin{array}{c}\Gamma'.\Xi.\varphi \text{ is trace composable} \\ \Gamma' \subseteq \Gamma \qquad \forall x \in \mathtt{dom}(\Gamma'), \Gamma'(x) \text{ is a base type} \\ \Delta' \subseteq \Delta \qquad \Xi;\Theta;\Sigma;u,\Gamma';\Delta' \vdash \varphi \ \textsf{silent} \\ \Xi,\Gamma' \vdash \varphi \ \textsf{ok} \qquad \Gamma|_{\mathtt{FAE}} \vdash e : \mathtt{FAE}\end{array}}{u;\Theta;\Sigma;\Gamma;\Delta \vdash e : \mathtt{inv}(\Xi.\varphi)} \ \textsc{Inv}$$

Invariant φ should be **trace composable**.

- $\Gamma.\Xi.\varphi$ is trace composable if given a substitution φ for Γ, three time points t_1, t_2, and t_3, such that $t_1 \leq t_2 \leq t_{3,} (\varphi(t_{1,} t_2)\gamma \wedge \varphi(t_{2,} t_3)\gamma) \Rightarrow \varphi(t_{1,} t_3)\gamma$.

Soundness

Soundness of System M’s type system is proved relative to the semantic interpretation model of computation types, expression types, and formula semantics.

Theorem 2 (Soundness).
If $\forall A :: \alpha \in \Sigma$, $\forall \mathcal{T}, \mathcal{K}, \iota, k, (k, A) \in \mathcal{RA}[\![\alpha]\!]^{\mathcal{K},\iota}_{\cdot;\mathcal{T}}$, *then*

1) $u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \tau$, $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall t, t'$, $t' \geq t$, *let* $\gamma_u = [t/u]$, $\forall \mathcal{T}$, $\forall k, \gamma$, $(k, \gamma) \in \mathcal{RG}[\![\Gamma\gamma_u]\!]^{t'}_{\theta;\mathcal{T}}$, $\mathcal{T} \vDash \Delta\gamma\gamma_u$ *implies* $(k, e\gamma) \in \mathcal{RE}[\![\tau\gamma\gamma_u]\!]^{t'}_{\theta;\mathcal{T}}$
2) $\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash c : \eta$, $\forall \mathcal{K}$, ι, *let* $\gamma_\Xi = [\mathcal{K}, \iota/\Xi, \mathsf{self}]$ $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \mathcal{T}$, $\forall k, \gamma, (k, \gamma) \in \mathcal{RG}[\![\Gamma\gamma_\Xi]\!]^{B(\mathcal{K})}_{\theta;\mathcal{T}}$, $\mathcal{T} \vDash \Delta\gamma\gamma_\Xi$ *implies* $(k, c\gamma) \in \mathcal{RC}[\![\eta\gamma\gamma_\Xi]\!]^{\mathcal{K},\iota}_{\theta;\mathcal{T}}$
3) $\Theta; \Sigma; \Gamma; \Delta \vdash \varphi$ true, $\forall t$ $\forall \theta \in \mathcal{RT}[\![\Theta]\!]$, $\forall \mathcal{T}$, $\forall k, \gamma, (k, \gamma) \in \mathcal{RG}[\![\Gamma]\!]^{t}_{\theta;\mathcal{T}}$, $\mathcal{T} \vDash \Delta\gamma$ *implies* $\mathcal{T} \vDash \varphi\gamma$

Composition and Rely-Guarantee Reasoning

Composition gives System M the ability to reason about distributed system with multiple running programs. System M uses a rely-guarantee style reasoning to compose local properties of programs running concurrently. **Rely-Guarantee** is a technique for reasoning about concurrent programs where programs can *rely* on the environment conforming to interference specifications, and the program is expected to guarantee that it conforms to its own interference specifications.

The following are three conditions for System M's rely-guarantee reasoning. $\psi(i,u)$ is the local guarantee of the thread *i* at time *u*. Predicate ζ is a set of threads that affect the state captured by the invariant φ.

$$
\begin{aligned}
&(\mathbf{RG}_1)\ \varphi(t_i)\\
&(\mathbf{RG}_2)\ \forall u, (\forall u', t_i < u' < u \Rightarrow \varphi(u')) \Rightarrow (\forall i, \zeta(i) \Rightarrow \psi(i, u))\\
&(\mathbf{RG}_3)\ \forall u, (\forall u', t_i < u' < u \Rightarrow \varphi(u')) \Rightarrow (\forall i, \zeta(i) \Rightarrow \psi(i, u))\\
&\qquad\qquad \Rightarrow \varphi(u)
\end{aligned}
$$

- RG1: invariant holds at t_i
- RG2: if invariant holds at all time points strictly less than *u*, then $\psi(i,u)$ holds for each *i* in ζ.
- RG3: If RG2 holds, then $\varphi(u)$ holds.

Example

Same counter example, but there are two threads. Initially, thread 1 has the lock. Would like to show that $\varphi(0,\infty)$ holds; that is, the counter never decreases.

$$
\begin{aligned}
F = \mathtt{fix}\ f(i) = \mathtt{comp}(&\mathtt{letc}\ x = \mathsf{download}();\\
&\mathtt{letc}\ y = \mathsf{check}\ x;\\
&y\ inc\ double\ prn;\\
&\mathsf{yieldTo}\ cnt\ i\\
&\mathtt{lete}\ \_ = f(i); \mathtt{ret}())\\
c_1 = \mathtt{lete}\ \_ = F\ \iota_2; \mathtt{ret}()&\\
c_2 = \mathtt{lete}\ \_ = F\ \iota_1; \mathtt{ret}()&
\end{aligned}
$$

Must prove that RG1, RG2, and RG3 holds.