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:

Base values  \( bv ::= tt \mid ff \mid \ell \mid n \mid ( ) \)

Expressions  \( e ::= x \mid bv \mid \lambda x.e \mid \text{fix } f(x).e \mid \Lambda X.e \mid e_1 \cdot e_2 \mid e \cdot \text{comp}(e) \)

Actions  \( a ::= A \mid a e \mid a \cdot \)

Computations  \( c ::= \text{act}(a) \mid \text{ret}(e) \mid \text{letc}(c_1, x.c_2) \mid \text{lete}(e_1, x.c_2) \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \)

- 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:

Stack  \( K ::= \[] \mid x.c :: K \)

Thread  \( T ::= \langle \nu ; K ; c \rangle \mid \langle \nu ; K ; e \rangle \mid \langle \nu ; \text{stuck} \rangle \)

Configuration  \( C ::= \sigma \triangleright T_1, \ldots, T_n \)

- Stack: a stack of frames. A frame is \( x.c \), where \( x \) binds the return expression of the computation preceding \( c \).
- Thread: A triple \( \langle \text{Thread_ID}, \text{Stack}, \text{Computation or Expression or Stuck} \rangle \)
  ○ Stuck means thread performed illegal action
- Configuration: shared state \( \sigma \) 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 $\sigma \triangleright T \leftrightarrow \sigma' \triangleright T'$. A sample of the transitions are provided below.

$$\sigma \triangleright T \leftrightarrow \sigma' \triangleright T'$$

\[ \begin{align*}
\text{next}(\sigma, a) &= (\sigma', e) \quad e \neq \text{stuck} \\
\sigma \triangleright (\langle \nu; x.c :: K; \mathbf{act}(a) \rangle) &\leftrightarrow \sigma' \triangleright (\langle \nu; K; c[e/x] \rangle) \\
\text{next}(\sigma, a) &= (\sigma', \text{stuck}) \\
\sigma \triangleright (\langle \nu; x.c :: K; \mathbf{act}(a) \rangle) &\leftrightarrow \sigma' \triangleright (\langle \nu; K; \text{stuck} \rangle) \\
\sigma \triangleright (\langle \nu; \text{stuck} \rangle) &\leftrightarrow \sigma \triangleright (\langle \nu; \text{stuck} \rangle) \\
\sigma \triangleright (\langle \nu; x.c :: K; \mathbf{ret}(e) \rangle) &\leftrightarrow \sigma \triangleright (\langle \nu; K; c[e/x] \rangle) \\
\sigma \triangleright (\langle \nu; K; \mathbf{let}(e_1, x, c_2) \rangle) &\leftrightarrow \sigma \triangleright (\langle \nu; x.c_2 :: K; e_1 \rangle) \\
\sigma \triangleright (\langle \nu; K; e \rangle) &\leftrightarrow \sigma \triangleright (\langle \nu; K; e' \rangle) \\
\sigma \triangleright (\langle \nu; K; \mathbf{comp}(c_1) \rangle) &\leftrightarrow \sigma \triangleright (\langle \nu; K; c_1 \rangle)
\end{align*} \]

- **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 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 $\text{cnt}$ can never decrease.

\[
\begin{align*}
\text{inc} &= \mathbf{comp}(\mathbf{letc} \quad x = \mathbf{read} \quad \text{cnt}; \mathbf{write} \quad \text{cnt} \ (x+1)) \\
\text{get} &= \mathbf{comp}(\mathbf{letc} \quad x = \mathbf{read} \quad \text{cnt}; \mathbf{ret}(x)) \\
\text{prn} &= \lambda y. \mathbf{comp}(\mathbf{letc} \quad _ = y \\
& \quad \quad \mathbf{letc} \quad x = \mathbf{read} \quad \text{cnt}; \mathbf{print} \quad x) \\
\text{c} &= \mathbf{letc} \quad x = \mathbf{download}(); \mathbf{letc} \quad y = \mathbf{check} \quad x; \\
& \quad \quad \mathbf{letc} \quad _ = y \ \text{inc} \ \text{get} \ \text{prn}; \mathbf{ret}()
\end{align*}
\]
• 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.

**Base types**

```
Base types  b ::= bool | nat | unit | ptr | time | thread
```

**Expr types**

```
Expr types  τ ::= X | b | Π x : τ₁ . τ₂ | ∀ X . τ | comp(ηₖ) 
                 | any | inv(Ξ . φ) | FAE
```

**Comp types**

```
Comp types  η ::= (x : τ , φ')
```

**Closed c types**

```
Closed c types  ηₖ ::= (Ξ , x : τ , φ₁ , Ξ , φ₂)
```

**Assertions**

```
Assertions  φ ::= P | e₁ = e₂ | φ e | ⊤ | ⊥ | ¬φ 
                 | φ₁ ∧ φ₂ | φ₁ ∨ φ₂ | ∀ x : τ . φ | ∃ x : τ . φ
```

**Expressions**

```
Expressions  e ::= · · · | self
```

**Exec ctx**

```
Exec ctx  Ξ ::= (uₐ : time , uₑ : time)
```

• 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 φ₁ 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 [uₐ, uₑ] .

```
φₙᵈ(uₐ , uₑ) = ∀ t₁ , t₂ , l , v₁ , v₂ , uₐ ≤ t₁ < t₂ ≤ uₑ ∧ eval cnt l 
               mem l v₁ t₁ ∧ mem l v₂ t₂ ⇒ v₂ ≥ v₁
```

• **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, \((<\text{step index}>, <\text{expression}>)\).
  - 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, \( \text{Type} \): 

\[
\text{Type} \overset{\text{def}}{=} \{ C \mid C \in \mathcal{P}(\{(j, \text{nf}) \mid j \in \mathbb{N}\}) \land \\
(\forall k, \text{nf}, (k, \text{nf}) \in C \land j < k \implies (j, \text{nf}) \in C) \}
\]

Interpretation of Computation Types

Interpretation of a computation type is denoted \( \mathcal{R}\mathcal{C}[\eta]^K,t \).

- This is a set of step-indexed computations, which are pairs \((k, c)\).
- \( \theta \) is a partial map from type variables to \( \text{Type} \).
- \( K=(t_b, t_e) \), is the time interval in which \( c \) executes.
- \( t \) is thread identifier.
- \( T \) is the trace.
- The following graphic shows an example trace \( T \) and the symbols in \( \mathcal{R}\mathcal{C}[\eta]^K,t \).

Interpretation of Expression Types

Two interpretations:

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

Formula Semantics

Formulas are interpreted over traces \( T \).

A sample of formula semantics is listed below.

- \( \varepsilon(T) \) is the set of atomic formulas that are true over \( T \).
• \textit{start}(e_1, \textit{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.

\[
\mathcal{T} \vdash P \ \bar{c} \ \text{iff} \ P \ \bar{c} \in \varepsilon(\mathcal{T}) \\
\mathcal{T} \vdash \text{start}(e_1, \text{comp}(c), e_2) \ \text{iff} \ e_1 \rightarrow^* \ i \ 
rightarrow^* \ e_2 \rightarrow^* \ t \ 
rightarrow^* \\
\text{and thread } i \text{ has } c \text{ as the active computation with an empty stack at time } t \text{ on } \mathcal{T} \\
\mathcal{T} \vdash \forall x: \tau. \varphi \ \text{iff} \ \forall e, (_, e) \in [\mathcal{T}] \ \text{implies} \ \mathcal{T} \vdash \varphi[e/x] \\
\mathcal{T} \vdash \exists x: \tau. \varphi \ \text{iff} \ \exists e, (_, e) \in [\mathcal{T}] \ \text{and} \ \mathcal{T} \vdash \varphi[e/x]
\]

\textbf{Type System and Assertion Logic}

Several environment contexts for typing judgements:

- \(\Theta\): type variables
- \(\Sigma\): specifications for action symbols
- \(\Gamma\): type bindings
- \(\Delta\): logical assertions
- \(\Xi\): compuation execution time interval
- \texttt{self}: the ID of the thread currently executing a computation

Examples of typing judgements are listed below.

- \(u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \tau\) \quad \text{expression } e \text{ has type } \tau
- \(\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash c : \eta\) \quad \text{computation } c \text{ has type } \eta
- \(\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash \varphi \text{ silent}\) \quad \varphi \text{ holds while reductions are non-effectful}
- \(\Theta; \Sigma; \Gamma; \Delta \vdash \varphi \text{ true}\) \quad \varphi \text{ 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 \((\Xi, \Gamma)\), then the invariant holds under non-effectful reductions/no reductions.

\[
\frac{\Theta; \Sigma; \Xi, \Gamma; \Delta \vdash \varphi \text{ true} \quad \Xi, \Gamma \vdash \varphi \text{ ok} \quad \text{SILENT}}{\Theta; \Sigma; \Xi; \Gamma; \Delta \vdash \varphi \text{ silent}}
\]

\textbf{Computation Typing}

Typing of computations, with the possible computations below.

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

A sampling of the ACT and RET typing rules are listed below.
• Act(…) draws actions from specifications from \( \Sigma \).
  ◦ 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.

\[
\frac{\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash \text{act}(a) : (x: \tau. \varphi_1[\Xi/\Xi'], \varphi_2[\Xi/\Xi'] \land \varphi)}{\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash a :: \text{Act}(\Xi'.x: \tau. \varphi_1, \Xi'.\varphi_2)}
\]  
\[\text{Act}\]

\[
\frac{E(\Xi); \Theta; \Sigma; B(\Xi); \Gamma; \Delta \vdash e : \tau}{\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash \text{ret}(e) : (x: \tau. ((x = e) \land \varphi), \varphi)}
\]  
\[\text{RET}\]

**Invariant \( \varphi \) should be trace composable.**

\[
\frac{\Gamma'.\Xi.\varphi \text{ is trace composable}}{\Gamma' \subseteq \Gamma \quad \forall x \in \text{dom}(\Gamma'), \Gamma'(x) \text{ is a base type}}
\]

\[
\frac{\Delta' \subseteq \Delta \quad \Xi; \Theta; \Sigma; u, \Gamma'; \Delta' \vdash \varphi \text{ silent}}{\Xi, \Gamma' / \varphi \text{ ok} \quad \Gamma|_{FAE} \vdash e : \text{FAE}}
\]

\[
\frac{\Xi; \Gamma' / \varphi \text{ ok} \quad \Gamma|_{FAE} \vdash e : \text{FAE}}{u; \Theta; \Sigma; \Gamma; \Delta \vdash e : \text{inv}(\Xi.\varphi)}
\]

**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 $\forall A :: \alpha \in \Sigma, \forall t, \mathcal{K}, \iota, k, (k, A) \in \mathcal{R}[\alpha]_{\mathcal{K}, t}^{\iota}, \text{then}$

1) $u; \Theta; \Sigma; \Gamma; \Delta \vdash c : \tau, \forall t \in \mathcal{R}[\Theta], \forall t, t', t' \geq t$, let $\gamma_u = [t/u], \forall k, \gamma, (k, \gamma) \in \mathcal{R}[\Gamma \gamma_u]_{\Theta; T}, T \vdash \Delta \gamma u$ implies $(k, e \gamma) \in \mathcal{R}[\gamma \gamma u]_{\Theta; T}$

2) $\Xi; \Theta; \Sigma; \Gamma; \Delta \vdash c : \eta, \forall K, \iota, \leq \gamma = [K, \iota/\Xi, \text{self}] \forall t \in \mathcal{R}[\Theta], \forall t, \gamma, (k, \gamma) \in \mathcal{R}[\Gamma \gamma]_{\Theta; T}, T \vdash \Delta \gamma \Xi$ implies $(k, c \gamma) \in \mathcal{R}[\gamma \gamma \Xi]_{\Theta; T}$

3) $\Theta; \Sigma; \Gamma; \Delta \vdash \varphi$ true, $\forall t \forall \theta \in \mathcal{R}[\Theta], \forall T, \forall k, \gamma, (k, \gamma) \in \mathcal{R}[\Gamma]_{\Theta; T}, T \vdash \Delta \gamma$ implies $T \vdash \varphi$

Composition and Rely-Guarantee Reasoning

Composition gives System M the ability to reason about distributed systems 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 $\zeta$ is a set of threads that affect the state captured by the invariant $\varphi$.

1. **RG1:** invariant holds at $t_i$
2. **RG2:** if invariant holds at all time points strictly less than $u$, then $\psi(i, u)$ holds for each $i$ in $\zeta$.
3. **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.

```plaintext
F = fix f(i) = comp(letc x = download();
      letc y = check x;
      y inc double prn;
      yieldTo cnt i
      letc _ = f(i); ret())

C_1 = letc _ = F t_2; ret()
C_2 = letc _ = F t_1; ret()
```
Must prove that RG1, RG2, and RG3 holds.