Rust Chapter3

An Introduction to Iris

K.Hirata

Iris による verification

- Iris の導入として、小さい example program を Iris で verify する。

Language

Language. For the purpose of this dissertation we instantiate Iris with HL (HeapLang): an ML-like language with higher-order store, fork, and compare-exchange ($\texttt{CmpX}$), as given below:

$$
\begin{aligned}
v, w \in \mathit{Val} &::= () \mid z \mid \texttt{true} \mid \texttt{false} \mid \ell \mid \lambda x.e \mid \qquad\qquad (\ell, z \in \mathbb{Z}) \\
&\quad\; (v, w) \mid \texttt{inl}(v) \mid \texttt{inr}(v) \\
e \in \mathit{Expr} &::= v \mid x \mid e_1(e_2) \mid \texttt{fork}\ \{e\} \mid \texttt{assert}(e) \mid \\
&\quad\; \texttt{ref}(e) \mid {!}\,e \mid e_1 \leftarrow e_2 \mid \texttt{CmpX}(e, e_1, e_2) \mid \ldots \\
K \in \mathit{Ctx} &::= \bullet \mid K(e) \mid v(K) \mid \texttt{assert}(K) \mid \\
&\quad\; \texttt{ref}(K) \mid {!}\,K \mid K \leftarrow e \mid v \leftarrow K \mid \\
&\quad\; \texttt{CmpX}(K, e_1, e_2) \mid \texttt{CmpX}(v, K, e_2) \mid \texttt{CmpX}(v, v_1, K) \mid \ldots
\end{aligned}
$$

We omit the usual projections on pairs and pattern matching on sums as well as primitive operations on values (comparison, addition, ...).

Operational semantics 1

- **Heap** σ は関数 $\{l$: locations$\} \to \{v$: values$\}$。

- **Head reduction** は、e_1 を heap σ_1 の状態で評価すると、e_2 へ step して、heap をσ_2 へ変化させ、新しいスレッドたち $\vec{e_f}$ を産む ($\vec{e_f}$ は expression の列)

Head reduction.

$$
\begin{array}{rll}
((\lambda x.e)(v), \sigma) \rightarrow_{\mathsf{h}} & (e[v/x], \sigma, \epsilon) & \\
(\mathbf{fork}\ \{e\}, \sigma) \rightarrow_{\mathsf{h}} & ((), \sigma, e) & \\
(\mathbf{assert}(\mathbf{true}), \sigma) \rightarrow_{\mathsf{h}} & ((), \sigma, \epsilon) & \\
(\mathbf{ref}(v), \sigma) \rightarrow_{\mathsf{h}} & (\ell, \sigma[\ell \leftarrow v], \epsilon) & \text{if } \sigma(\ell) = \bot \\
(!\,\ell, \sigma[\ell \leftarrow v]) \rightarrow_{\mathsf{h}} & (v, \sigma[\ell \leftarrow v], \epsilon) & \\
(\ell \leftarrow w, \sigma[\ell \leftarrow v]) \rightarrow_{\mathsf{h}} & ((), \sigma[\ell \leftarrow w], \epsilon) & \\
(\mathbf{CmpX}(\ell, w_1, w_2), \sigma[\ell \leftarrow v]) \rightarrow_{\mathsf{h}} & ((v, \mathbf{true}), \sigma[\ell \leftarrow w_2], \epsilon) & \text{if } v \cong w_1 \\
(\mathbf{CmpX}(\ell, w_1, w_2), \sigma[\ell \leftarrow v]) \rightarrow_{\mathsf{h}} & ((v, \mathbf{false}), \sigma[\ell \leftarrow v], \epsilon) & \text{if } v \ncong w_1
\end{array}
$$

If $!l == w_1 \{ l \leftarrow w_2 \}$

カンニング

The operational semantics are defined in Figure 3.1. The state of a program execution is given by a *heap* σ mapping locations ℓ to values v, and a *thread pool* storing the expression each thread is executing. *Thread-pool reduction* $(T_1, \sigma_1) \rightarrow_{\mathsf{tp}} (T_2, \sigma_2)$ just picks some thread non-deterministically[2] and takes a step in that thread. *Thread reduction* uses evaluation contexts K to find a reducible *head expression*[3] and reduces it. *Head reduction* $(e_1, \sigma_1) \rightarrow_{\mathsf{h}} (e_2, \sigma_2, \vec{e}_f)$ says that head expression e_1 with current heap σ_1 can step to e_2, change the heap to σ_2 and spawn new threads $\vec{e}_f$ (this is a *list* of expressions) in the process. We use "stuckness" (irreducible expressions) to model bogus executions, like a program that tries to use a Boolean (or an integer) to access memory, or a program that runs into **assert**(**false**).

Operational semantics 2

- **Thread pool** $T = e_1; e_2; \ldots; e_n$ は それぞれの thread が実行している expression をためておく列。

- **Thread-pool reduction** (右) $(T_1, \sigma_1) \to_{\mathsf{tp}} (T_2, \sigma_2)$ は Thread pool から1つ**非決定的に**スレッドを取り出して、1 step 実行するもの。

- **Thread reduction** は、reducible な head expression を表すものを取り出す。(矢印の添字に注目。) K は evaluation context。 これは決定的。

Thread-local and threadpool reduction.

$$\frac{(e,\sigma) \to_{\mathsf{h}} (e',\sigma',\vec{e}_f)}{(K[\,e\,],\sigma) \to_{\mathsf{t}} (K[\,e'\,],\sigma',\vec{e}_f)} \qquad \frac{(e,\sigma) \to_{\mathsf{t}} (e',\sigma',\vec{e}_f)}{(T_1;e;T_2,\sigma) \to_{\mathsf{tp}} (T_1;e';T_2;\vec{e}_f,\sigma')}$$

- (K 再掲)

$$\begin{aligned} K \in \mathit{Ctx} ::= {} & \bullet \mid K(e) \mid v(K) \mid \mathtt{assert}(K) \mid \\ & \mathtt{ref}(K) \mid\, !K \mid K \leftarrow e \mid v \leftarrow K \mid \\ & \mathtt{CmpX}(K, e_1, e_2) \mid \mathtt{CmpX}(v, K, e_2) \mid \mathtt{CmpX}(v, v_1, K) \mid \ldots \end{aligned}$$

- We use “stuckness” (irreducible expressions) to model bogus executions, like a program that tries to use a Boolean (or an integer) to access memory, or a program that runs into assert(false).
- $\cong$ は partial な equality (型が異なるものに対しては定義されていない etc.) $\ncong$ は比較可能で異なる値をあらわす。比較可能でないときは stuck.

Logic.

- 高階分離論理の文法 (今はよく分からなくていい)

$$P, Q, R ::= \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid$$
$$P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v \mid t = u \mid$$
$$\Box P \mid \boxed{P}^{\mathcal{N}} \mid \boxed{a}^{\gamma} \mid \mathcal{V}(a) \mid \{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} \mid P \Rrightarrow_{\mathcal{E}} Q \mid \ldots$$

- Some of the proof rules for Hoare triple

HOARE-FRAME
$$\frac{\{P\}\, e\, \{w.\, Q\}_{\mathcal{E}}}{\{P * R\}\, e\, \{w.\, Q * R\}_{\mathcal{E}}}$$

HOARE-VALUE
$$\{\mathsf{True}\}\, v\, \{w.\, w = v\}_{\mathcal{E}}$$

HOARE-BIND
$$\frac{\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} \qquad \forall v.\, \{Q\}\, K[v]\, \{w.\, R\}_{\mathcal{E}}}{\{P\}\, K[e]\, \{w.\, R\}_{\mathcal{E}}}$$

HOARE-λ
$$\frac{\{P\}\, e[v/x]\, \{w.\, Q\}_{\mathcal{E}}}{\{P\}\, (\lambda x.e)(v)\, \{w.\, Q\}_{\mathcal{E}}}$$

HOARE-FORK
$$\frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P\}\, \mathbf{fork}\ \{e\}\ \{\mathsf{True}\}_{\mathcal{E}}}$$

HOARE-ASSERT
$$\{\mathsf{True}\}\, \mathbf{assert}(\mathbf{true})\, \{\mathsf{True}\}_{\mathcal{E}}$$

HOARE-ALLOC
$$\{\mathsf{True}\}\, \mathbf{ref}(v)\, \{\ell.\, \ell \mapsto v\}_{\mathcal{E}}$$

HOARE-LOAD
$$\{\ell \mapsto v\}\ !\ell\, \{w.\, w = v * \ell \mapsto v\}_{\mathcal{E}}$$

HOARE-STORE
$$\{\ell \mapsto v\}\, \ell \leftarrow w\, \{\ell \mapsto w\}_{\mathcal{E}}$$

HOARE-CMPX-SUC
$$\frac{v \cong w_1}{\{\ell \mapsto v\}\, \mathbf{CmpX}(\ell, w_1, w_2)\, \{v'.\, v' = (v, \mathbf{true}) * \ell \mapsto w_2\}_{\mathcal{E}}}$$

HOARE-CMPX-FAIL
$$\frac{v \ncong w_1}{\{\ell \mapsto v\}\, \mathbf{CmpX}(\ell, w_1, w_2)\, \{v'.\, v' = (v, \mathbf{false}) * \ell \mapsto v\}_{\mathcal{E}}}$$

HEAP-EXCLUSIVE
$$\ell \mapsto v * \ell \mapsto w \vdash \mathsf{False}$$

- Hoare triple $\{P\}\, e\, \{v\,.\, Q\}$ の v は e の return value を表す。

- Bind rule は、chain statements の sequence rule の一般化。(e を verify して v を得て、v を使って context K を verify)

- Hoare-Fork は現在のスレッドが所有する任意の resource P は、forked-off されたスレッドで使用可能であることを意味している。

$$\frac{\{P\}\, e\, \{\mathsf{True}\}}{\{P * Q\}\ \mathtt{fork}\ \{e\}\ \{Q\}_{\mathcal{E}}}$$

- こう書くと、resource P は手放して子に与え、Q は current thread が持つ、的な意味にとれる。

- 子から resource を取り戻すものは Iris の built-in にはないが、Hoare Logic 内で implement して Iris で verify することは可能 (一つ内側の世界)

- quantifier は高階論理の quantifier。すなわち、「任意の論理式に対して」が可能。

The motivating example

Example code.

$$
\begin{aligned}
&\texttt{mk\_oneshot} := \lambda\_. \\
&\quad \textbf{let } x = \textbf{ref}(\textbf{inl}(0)); \\
&\quad \{ \ \texttt{set} = \lambda n.\, \textbf{let } (\_, b) = \texttt{CmpX}(x, \textbf{inl}(0), \textbf{inr}(n)); \\
&\qquad\qquad\qquad \texttt{assert}(b), \\
&\quad\ \ \texttt{check} = \lambda\_.\, \textbf{let } y = !\, x; \\
&\qquad\qquad\qquad \lambda\_.\, \textbf{let } y' = !\, x; \\
&\qquad\qquad\qquad\quad \textbf{match } y \textbf{ with} \\
&\qquad\qquad\qquad\quad\ \ \textbf{inl}(\_) \Rightarrow () \\
&\qquad\qquad\qquad\quad \mid \textbf{inr}(\_) \Rightarrow \texttt{assert}(y = y') \\
&\qquad\qquad\qquad\quad \textbf{end} \quad \}
\end{aligned}
$$

mk_oneshot: x を allocate して、クロージャの組を返す。

set: x に inr(n) を set する。元々は inl(0) (初期値) が入っていたことを assert。

check: まず x に含まれる値を記録してクロージャを返す。そのクロージャが呼ばれたとき、x の中身が inr(_) だったなら変わっていないことを assert.

Example code.

$$
\begin{array}{l}
\texttt{mk\_oneshot} := \lambda\_. \\
\quad \texttt{let}\ x = \texttt{ref}(\texttt{inl}(0)); \\
\quad \{\ \ \texttt{set} = \lambda n.\ \texttt{let}\ (\_, b) = \texttt{CmpX}(x, \texttt{inl}(0), \texttt{inr}(n)); \\
\qquad\qquad\quad \texttt{assert}(b), \\
\quad\ \ \texttt{check} = \lambda\_.\ \texttt{let}\ y = !\,x; \\
\qquad\qquad\quad \lambda\_.\ \texttt{let}\ y' = !\,x; \\
\qquad\qquad\qquad \texttt{match}\ y\ \texttt{with} \\
\qquad\qquad\qquad\ \ \texttt{inl}(\_) \Rightarrow () \\
\qquad\qquad\qquad \mid \texttt{inr}(\_) \Rightarrow \texttt{assert}(y = y') \\
\qquad\qquad\qquad \texttt{end}\ \ \}
\end{array}
$$

x は外から見えないので、
x の中身は初期値 inl(0) か
set で更新された inr(n) しか
取り得ない。check は更新しない。

いま、set は呼ばれた時に中身が inl(0) で
あることを assert するので、2 回呼ばれる
と stuck する。

check の中の assertion が呼ばれるとき、クロージャが生成され
たときに既に set が1回呼ばれていて、さらに生成されてからは呼
ばれていないはずなので、このassertion は fail しない。

Specifying the example.

Example specification.

$$\{\mathsf{True}\}$$
$$\mathtt{mk\_oneshot}()$$
$$\left\{\begin{array}{l} c.\ \exists T.\, T * \big(\forall v.\ \{T\}\ c.\mathtt{set}(v)\ \{\mathsf{True}\}\big) * \\ \quad \{\mathsf{True}\}\ c.\mathtt{check}()\ \{f.\ \{\mathsf{True}\}\ f()\ \{\mathsf{True}\}\} \end{array}\right\}$$

- T が resource (ownership) で、set は T を消費する。最初は T を一つ持っている。T は複製できない。
- check はいつでも呼ぶことができて、f を返す。f はいつも呼び出せる (何もしない)。
- “クロージャを返す” というのは、Hoare がネストされていることで表現されている。
- Iris では複製はできないが drop はできるので、これは **affine separation logic.** ($P * Q \Rightarrow P$ は導ける、の意味。) ちなみに $P * \mathsf{True} \Leftrightarrow P$.

High-level proof structure

- さっきの証明をしようと思うと、x が「ただ一度だけ update できる」ことを encode しないといけない。
- そのために Ghost location $\boxed{a}^{\gamma}$ を allocate する。(name: γ、value: a)
- Ghost state は x の current state をそのまま反映する。
  (Iris の ghost location l ↔ 分離論理の (physical) location γ)
- Ghost state によって、どのような sharing がその location で可能かどうかを指定する。
  - Physical location の $l \mapsto v$ は、l の full ownership を表していたが、ghost location γ にどんな structure と所有権を許すかは、選ぶことができる (?)
  - of it). In contrast, Iris permits us to choose whatever kind of structure and ownership we want for our ghost location γ; in particular, we can define it in such a way that, although the contents of γ mirror the contents of x, we can freely share ownership of γ once it has been initialized (by a call to **set**). This in turn will allow the closure returned by **check** to own a piece of γ witnessing its value after initialization. We will then have an *invariant* (see §3.2) tying the value of γ to the value of x, so we *know* which value that closure is going to see when it reads from x, and we know that that value is going to match y.

Another way to describe what is happening is to say that we are applying the idea of fictional separation: The separation on (after set was called) is “fictional” in the sense that multiple threads can own parts of and therefore manipulate the same shared variable x, the two being tied together by an invariant.