Iris-OS Documentation

Zhen Zhang

March 26, 2017

Abstract

This document describes formally the OS verification framework based on the Iris program logic. The latest versions of Iris document and the Iris Coq formalization can be found in the git repository at `https://gitlab.mpi-sws.org/FP/iris-coq/`.

Contents

1 Language

1.1 Definitions

Syntax. The language is a simplified version of C. It consists of *Prim* (primitives), *Expr* (expressions), *Type* (types) and *Val* (values).

Memory address $l : \mathit{Addr} \triangleq b\#o$ where block address $b \in \mathbb{N}$, offset $o \in \mathbb{Z}^+$.

$$\begin{aligned}
\tau : \mathit{Type} &::= \tau_{\texttt{void}} \mid \tau_{\texttt{null}} \mid \tau_{\texttt{int8}} \mid \tau_{\texttt{int32}} \mid *\tau \mid \tau \times \tau \\
v : \mathit{Val} &::= \texttt{void} \mid \texttt{null} \mid i \in [0, 2^8) \mid i \in [0, 2^{32}) \mid l \mid (v, v) \\
p : \mathit{Prim} &::= \texttt{cli} \mid \texttt{sti} \\
e : \mathit{Expr} &::= v \mid x \mid e \oplus e \mid !\, e \mid !_\tau\, e \mid \&e \mid e.1 \mid e.2 \\
&\quad\ \texttt{skip} \mid p \mid e \leftarrow e \mid \texttt{if}(e)\,\{e\}\,\texttt{else}\,\{e\} \mid \texttt{while}_e(e)\,\{e\} \mid \\
&\quad\ \texttt{return}\, e \mid f(e_1, ..., e_n) \mid e; e \mid \texttt{alloc}_\tau(e)
\end{aligned}$$

The relationship between $!\, e$ and $!_\tau\, e$: The former one is in source, and the later one is "instantiated" with type information and used in inference rules. It is part of the efforts of removing lexical analysis from core program logic.

Note: the C program is usually more restrictive by differentiating statements and expressions. For simplicity and expressivity, we merged them together into a coherent expression *Expr* without loss of generality.

Program. A program is considered to be a set of functions *Function*, each identified by its name. Each function is a triple of return type τ_{ret}, parameter declarations $(x_1 : \tau_1, ...)$, and function body e.

Evaluation Context. To make the evaluation order explicit and reusable by the WP-BIND rule, we define evaluation context.

$$\begin{aligned}
K : \textsc{Ectx} \triangleq\ & \bullet \mid \bullet \oplus e \mid v \oplus \bullet \mid !\,\bullet \mid \&\bullet \mid \bullet.1 \mid \bullet.2 \mid \\
& \bullet \leftarrow e \mid l \leftarrow \bullet \mid \texttt{if}(\bullet)\,\{e\}\,\texttt{else}\,\{e\} \mid \texttt{while}_e(\bullet)\,\{e\} \mid \\
& \texttt{return}\,\bullet \mid f(v_1, ..., \bullet, e_1, ...) \mid \texttt{alloc}_\tau(\bullet) \mid \bullet; e
\end{aligned}$$

1.2 Semantics

Model. Define continuation $\textsc{Cont} \triangleq [K]$, and stack $\textsc{Stack} \triangleq [\textsc{Cont}]$.

We define byte-size value and memory model:

$$v_{\text{ byte}} : \mathit{Val}_{\text{ byte}} \triangleq \lightning \mid i \in [0, 2^8) \mid l_{\{0|1|2|3\}} \mid \texttt{null}$$

We define state $\sigma : \mathit{State} \triangleq (\text{heap} : [l \hookrightarrow v_{\text{ byte}}], \text{text} : [x \hookrightarrow \mathit{Function}], \text{stack} : \textsc{Stack})$,

You may notice the difference between *Val* and $\mathit{Val}_{\text{ byte}}$, it is intentional to create a layer of abstraction for easier manipulation of spatial assertions and also a clean, unified language syntax.

ES-BINOP
$$\frac{[\![oplus]\!](v_1, v_2) = v'}{(v_1 \oplus v_2, \sigma) \rightarrow_{Expr} (v', \sigma)}$$

ES-DEREF-TYPED
$$\frac{\vdash_{\text{tychk}} v : \tau \quad \sigma \vdash l \mapsto \text{encode}(v)}{(\,!_{\tau}\, l, \sigma) \rightarrow_{Expr} (v, \sigma)}$$

ES-FST
$$((v_1, v_2).1, \sigma) \rightarrow_{Expr} (v_1, \sigma)$$

ES-SND
$$((v_1, v_2).2, \sigma) \rightarrow_{Expr} (v_2, \sigma)$$

ES-ASSIGN
$$(l \leftarrow v, \sigma) \rightarrow_{Expr} (\texttt{skip}, \sigma[l \mapsto \text{encode}(v)])$$

ES-SEQ
$$(v; e, \sigma) \rightarrow_{Expr} (e, \sigma)$$

ES-ALLOC
$$\frac{\vdash_{\text{tychk}} v : \tau \quad \forall o'.\, \sigma(b, o') = \bot}{(\texttt{alloc}_{\tau}(v), \sigma) \rightarrow_{Expr} (b\#o, \sigma[b\#o \mapsto v])}$$

ES-WHILE-TRUE
$$(\texttt{while}_e(\mathsf{true})\, \{s\}, \sigma) \rightarrow_{Expr} (s; \texttt{while}_e(e)\, \{s\}, \sigma)$$

ES-WHILE-FALSE
$$(\texttt{while}_e(\mathsf{false})\, \{s\}, \sigma) \rightarrow_{Expr} (\texttt{skip}, \sigma)$$

ES-BIND'
$$\frac{\text{is\_jump}(e) = \mathsf{False} \quad (e, \sigma) \rightarrow_{Expr} (e', \sigma')}{((k :: ks)e, \sigma) \rightarrow_{Expr} ((k :: ks)e', \sigma')}$$

Figure 1: Semantics rules

Small-Step Operational Semantics. We define the HNF small step semantics for both non-jumping expression $((e, \sigma) \rightarrow_{Expr} (e', \sigma'))$ and jumping expressions $((e, s : \text{STACK}) \rightarrow^{\sigma}_{jump} (e', s' : \text{STACK}))$, and then combine them together $((e, \sigma, s) \rightarrow_{\text{cur}} (e', \sigma', s'))$.

XXX: Define $\sigma \vdash l \mapsto v\ \vec{\text{byte}}$

1.3 Type System and Environment

Local Typing Rules. The types are defined in §1.1, and all values in v can be *locally* typed trivially (since v is introduced to reflect type structure in some sense). Nevertheless, due to the fact that language of study is weakly-typed in the vein of C, we still have some "weird" rules worth documenting.

Here we define local typing judgment $\vdash_{\text{tychk}} v : \tau$ for values.

TYCHK-VOID
$$\vdash_{\text{tychk}} \texttt{void} : \tau_{\texttt{void}}$$

TYCHK-NULL
$$\vdash_{\text{tychk}} \texttt{null} : \tau_{\texttt{null}}$$

TYCHK-INT8
$$\vdash_{\text{tychk}} i \in [0, 2^8) : \tau_{\texttt{int8}}$$

TYCHK-INT32
$$\vdash_{\text{tychk}} i \in [0, 2^{32}) : \tau_{\texttt{int32}}$$

TYCHK-NULL-PTR
$$\forall \tau.\ \vdash_{\text{tychk}} \texttt{null} : *\tau$$

TYCHK-PTR
$$\forall \tau, l.\ \vdash_{\text{tychk}} l : *\tau$$

TYCHK-PROD
$$\frac{\vdash_{\text{tychk}} v_1 : \tau_1 \quad \vdash_{\text{tychk}} v_2 : \tau_2}{\vdash_{\text{tychk}} (v_1, v_2) : \tau_1 \times \tau_2}$$

Typing Environment When variables are introduced, we will need an environment $\Gamma : \mathit{Env} \triangleq [x \hookrightarrow (\tau, l)]$ to "unfold" the meaning of variables.

In Iris-OS, variables are all unfolded before the running the function body, which saves the program logic from caring about the lexical environment. During this unfolding, we will replace variables with either their location (left-hand side) or the dereference of their location (right-hand side). And we will also produce a pointer arithmetic expression when processing the left-hand side expression, which requires type inference $\vdash_{\text{tyinf}} e : \tau$ (since it is standard and trivial, we will leave out the details here).

Then we define the rules for "interpreting" left-hand side expression ($(\!|e|\!)_{\text{LHS}}$) and right-hand side expression ($(\!|e|\!)_{\text{RHS}}$). In all rules below, a Γ is implicitly captured, and if any operation on Γ failed, then the rule will be returning "invalid" in implementation.

$$\begin{aligned}
(\!|e_1 \leftarrow e_2|\!)_{\text{LHS}} &= (\!|e_1|\!)_{\text{LHS}} \leftarrow e_2 \\
(\!|x|\!)_{\text{LHS}} &= \Gamma(x).l \\
(\!|\,!\,e|\!)_{\text{LHS}} &= (\!|e|\!)_{\text{RHS}} \\
(\!|e.1|\!)_{\text{LHS}} &= (\!|e|\!)_{\text{LHS}} \\
(\!|e.2|\!)_{\text{LHS}} &= (\!|e|\!)_{\text{LHS}} + \text{sizeof}(\tau) \quad \text{if } \vdash_{\text{tyinf}} (\!|e|\!)_{\text{LHS}} : *(\tau_1 \times \tau_2) \\
(\!|l|\!)_{\text{LHS}} &= l
\end{aligned}$$

$$\begin{aligned}
(\!|e_1 \leftarrow e_2|\!)_{\text{RHS}} &= e_1 \leftarrow (\!|e_2|\!)_{\text{RHS}} \\
(\!|x|\!)_{\text{RHS}} &= !_{\Gamma(x).t}\,\Gamma(x).l \\
(\!|\,!\,e|\!)_{\text{RHS}} &= !_{\tau}\,(\!|e|\!)_{\text{RHS}} \quad \text{if } \vdash_{\text{tyinf}} e : *\tau
\end{aligned}$$

The cases not covered are defined recursively (and trivially)

2 Program Logic

This section describes how to build a program logic for the C language (*c.f.* §1) on top of the base logic of Iris.

2.1 Extended Assertions

Using the standard Iris assertions and the ownership of ghost heap resource, we can define some basic custom assertions:

$$\begin{aligned} l \mapsto_q v : t &\triangleq l \mapsto_q \text{encode}(v) \wedge \vdash_{\text{tychk}} v : t \\ l \mapsto v : t &\triangleq l \mapsto_1 v : t \\ l \mapsto_q - : t &\triangleq \exists v.\, l \mapsto_q v : t \end{aligned}$$

2.2 Weakest Precondition

Finally, we can define the core piece of the program logic, the assertion that reasons about program behavior: Weakest precondition, from which Hoare triples will be derived.

Defining weakest precondition. While we have fixed the program state σ in language definition, but it can be any state, as long as it has a predicate $S : \mathit{State} \to \mathit{iProp}$ that interprets the physical state as an Iris assertion. For our heap state,

$$S(\sigma) \triangleq \mathsf{Own}(\bullet\, \text{fmap}(\lambda v.(1, \mathsf{ag}v), \sigma.\text{heap})) * \mathsf{Own}(\bullet\, \sigma.\text{text}) * \mathsf{Own}((\frac{1}{2}, \sigma.\text{stack}))$$

$$\begin{aligned} \mathit{wp} \triangleq\ & \mu\, \mathit{wp}.\, \lambda \mathcal{E}, E_{\text{cur}}, \Phi, \Phi_{\text{ret}}. \\ & \quad (\exists v.\, \text{to\_val}(E_{\text{cur}}) = v \wedge \mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} \Phi(v)) \vee \\ & \quad (\exists v.\, \text{to\_ret\_val}(E_{\text{cur}}) = v \wedge \mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} \Phi_{\text{ret}}(v)) \vee \\ & \quad \Big(\text{to\_val}(E_{\text{cur}}) = \bot \wedge \text{to\_ret\_val}(E_{\text{cur}}) = \bot \wedge \\ & \qquad \forall \sigma.\, S(\sigma) \;{}^{\mathcal{E}}\!\!\Rrightarrow\!\!\ast^{\emptyset} \\ & \qquad \text{red}(e, \sigma) * \triangleright \forall E'_{\text{cur}}, \sigma'.\, (E_{\text{cur}}, \sigma \to E'_{\text{cur}}, \sigma') \;{}^{\emptyset}\!\!\Rrightarrow\!\!\ast^{\mathcal{E}} \\ & \qquad\quad S(\sigma') * \mathit{wp}(\mathcal{E}, E'_{\text{cur}}, \Phi, \Phi_{\text{ret}})\Big) \end{aligned}$$

Here are some conventions:

- If we leave away the mask, we assume it to default to $\top$.
- We will leave † out when writing E_{cur} in WP.
- Φ in post-condition might or might not take a value parameter, depending on the context.

Laws of weakest precondition. The following rules can all be derived:

WP-VALUE
$$\Phi(v) \vdash \mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi\}$$

WP-SKIP
$$\triangleright\, \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, v; e\, \{\Phi\}$$

WP-RET
$$\text{stack}(k' :: ks) * (\text{stack}(ks) \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, k'(v)\, \{\Phi\}) \vdash \mathsf{wp}_{\mathcal{E}}\, k(\texttt{return}\, v)\, \{\Phi\}$$

WP-STRONG-MONO
$$\frac{\mathcal{E}_1 \subseteq \mathcal{E}_2}{((\forall v.\, \Phi(v) \Rrightarrow_{\mathcal{E}_2} \Psi(v)) \wedge (\forall v.\, \Phi_{\text{ret}}(v) \Rrightarrow_{\mathcal{E}_2} \Psi_{\text{ret}}(v))) * \mathsf{wp}_{\mathcal{E}_1}\, E_{\text{cur}}\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}_2}\, E_{\text{cur}}\, \{\Psi\} \Psi_{\text{ret}}}$$

FUP-WP
$$\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, E_{\text{cur}}\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, E_{\text{cur}}\, \{\Phi\}$$

WP-FUP
$$\mathsf{wp}_{\mathcal{E}}\, E_{\text{cur}}\, \{x.\, \Rrightarrow_{\mathcal{E}} \Phi(x)\} x.\, \Rrightarrow_{\mathcal{E}} \Phi_{\text{ret}}(x) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

WP-BIND
$$\frac{\text{is\_jump}(e) = \mathsf{False}}{\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \mathsf{wp}_{\mathcal{E}}\, k(x)\, \{\Phi\}\} \vdash \mathsf{wp}_{\mathcal{E}}\, k(e)\, \{\Phi\}}$$

WP-OP
$$\frac{[\![oplus]\!](v_1, v_2) = v'}{\Phi(v') \vdash \mathsf{wp}_{\mathcal{E}}\, v_1 \oplus v_2\, \{\Phi\}}$$

WP-ASSIGN
$$\frac{\vdash_{\text{tychk}} v : \tau' \qquad \surd(\tau \leftarrow \tau')}{l \mapsto - : \tau * (l \mapsto v : \tau \mathrel{-\!\!*} \Phi) \vdash \mathsf{wp}_{\mathcal{E}}\, l \leftarrow v\, \{\Phi\}}$$

WP-LOAD
$$l \mapsto_q v : \tau * (l \mapsto_q v : \tau \mathrel{-\!\!*} \Phi(v)) \vdash \mathsf{wp}_{\mathcal{E}}\, !_{\tau}\, l\, \{\Phi\}$$

WP-SEQ
$$\frac{\text{is\_jump}(e_1) = \mathsf{False}}{\mathsf{wp}_{\mathcal{E}}\, e_1\, \{v,\, \mathsf{wp}_{\mathcal{E}}\, v; e_2\, \{\Phi\}\} \vdash \mathsf{wp}_{\mathcal{E}}\, e_1; e_2\, \{\Phi\}}$$

WP-WHILE-TRUE
$$\frac{\mathsf{wp}_{\mathcal{E}}\, s; \texttt{while}_e(e)\, \{s\}\, \{\Phi\}}{\mathsf{wp}_{\mathcal{E}}\, \texttt{while}_e(\mathsf{true})\, \{s\}\, \{\Phi\}}$$

WP-WHILE-FALSE
$$\frac{\Phi(\texttt{void})}{\mathsf{wp}_{\mathcal{E}}\, \texttt{while}_e(\mathsf{false})\, \{s\}\, \{\Phi\}}$$

WP-WHILE-INV
$$\frac{\begin{array}{c}\forall \Phi.\, (I * (\forall v.\, (v = \mathsf{false} * Q(\texttt{void})) \vee (v = \mathsf{true} * I)) \mathrel{-\!\!*} \Phi(v)) \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Phi\} \\ \forall \Phi.\, (I * (I \mathrel{-\!\!*} \Phi(\texttt{void}))) \mathrel{-\!\!*} \mathsf{wp}\, s\, \{\Phi\}\end{array}}{I \vdash \mathsf{wp}\, \texttt{while}_e(e)\, \{s\}\, \{Q\}}$$

WP-FST
$$\triangleright\, \Phi(v_1) \vdash \mathsf{wp}_{\mathcal{E}}\, (v_1, v_2).1\, \{\Phi\}$$

WP-SND
$$\triangleright\, \Phi(v_2) \vdash \mathsf{wp}_{\mathcal{E}}\, (v_1, v_2).1\, \{\Phi\}$$

WP-ALLOC
$$\frac{\vdash_{\text{tychk}} v : \tau}{(\forall l.\, l \mapsto v : \tau \mathrel{-\!\!*} \Phi(l)) \vdash \mathsf{wp}_{\mathcal{E}}\, \texttt{alloc}_{\tau}(v)\, \{\Phi\}}$$

WP-CALL
$$f \mapsto_{\text{text}} (ps, e) * \text{stack}(ks) * \triangleright(\text{stack}(k :: ks) \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, \text{instantiate}(ps, ls, e)\, \{\Phi\}) \vdash \mathsf{wp}_{\mathcal{E}}\, k(f(vs))\, \{\Phi\}$$

3 Misc

This section contained some key formal developments claimed but not yet mechanized in Coq for reference.

3.1 Properties of Evaluation Context

Our expression *Expr* is a recursively defined algebraic data type, which can be generalized into the following form:

$$e : \mathit{Expr} ::= \mathit{Expr}_1(A_1, e^{r_1}) \mid ... \mid \mathit{Expr}_n(A_n, e^{r_n}) \mid v$$

Here, Expr_i is tag for i-th class of expression, A_i is its arbitrary non-recursive payload, and e^{r_i} means it has $r_i \in \mathbb{N}$ recursive occurences.

It is apparent that Ectx is a direct translation of *Expr* plus some ordering considerations, though in actual Coq development we don't mechanize this fact. But we will make it explicit here to support some stronger claims than what we can do in Coq.

For some abstract *Expr* like above, its Ectx should be a sum of sub-Ectx$_i$ for each Expr_i. When $r_i = 0$, Ectx$_i$ doesn't exist, now we consider $r_i > 0$, define

$$k_i : \textsc{Ectx}_i ::= \textsc{Ectx}_{i1}(A_i, e^{r_i - 1}) \mid \textsc{Ectx}_{i2}(A_i, v^1, e^{r_i - 2}) \mid ... \mid \textsc{Ectx}_{ir_i}(A_i, v^{r_i - 1})$$

Theorem 3.1. *For any $e, e' : \mathit{Expr}$ and $k_{im}, k_{jn} : \textsc{Ectx}$,*

$$k_{im}(e) = k_{jn}(e') \vdash i = j$$

Proof. Trivial. □

Theorem 3.2. *For any $e, e' : \mathit{Expr}$ and $k_{im}, k_{in} : \textsc{Ectx}_i$,*

$$k_{im}(e) = k_{in}(e') \vdash (m = n \land e = e') \lor (m \neq n \land (\exists v. \text{to\_val}(e) = v \lor \exists v. \text{to\_val}(e) = v))$$

Proof. When $m = n$, by injectivity; When $m \neq n$, we can expand the equation like below without loss of generality:

$$\begin{aligned}&\mathit{Expr}_i(A_i, v_1, ..., v_{m-1}, e, e_{m+1}, ..., e_{r_i - 1}) = \\ &\mathit{Expr}_i(A'_i, v'_1, ..., v'_{m-1}, v'_m, ..., e', ...)\end{aligned}$$

So $e = v'_m$ by injectivity. □

Now, if we partition *Expr* into complete groups of disjoint classes, like E_v, E_{red}, E_{jmp}, then we can trivially derive following lemma:

Lemma 3.3. *For any $e, e' : \mathit{Expr}$ and $k, k' : \textsc{Ectx}$,*

$$k(e) = k'(e') \vdash (k = k' \land e = e') \lor (k \neq k' \land (e \in E_v \lor e' \in E_v))$$

Lemma 3.4. *For any $e \in E_{red}, e' \in E_{jmp}$ and $k, k' : \textsc{Ectx}$,*

$$k(e) = k'(e') \vdash k = k' \land e = e'$$

Now we define continuation $K : \textsc{Cont} \triangleq [\textsc{Ectx}]$, and corresponding fill operation as a trivial fold. You can imagine an arbitrary expression as a tree, in which the nodes might be either evaluated (the leaves), in some normal form ready to be evaluated (leaves' parent), or not in normal form at all. So for an expression e, the intuition might be there is only an *unique* way of extracting it into a $K(e')$, in which e' is either a value or in some normal form. This implies a very general injectivitity lemma for continuation:

Lemma 3.5. *If e, e' is in some normal form, and $K(e) = K'(e')$, then $K = K' \wedge e = e'$.*

Proof. We can prove it inductively with 3.3.

- If both K, K' is empty, then it is trivially proven.
- If only one of K, K', say K, is empty, then we have $e = K''(k'(e'))$ as assumption, which leads to contradiction.
- Now we can rewrite K as $k :: K_1$ for some k, and K' as $k' :: K_2$, it is apparent that either $K_1(e)$ or $K_2(e)$ can be in evaluated form, so by 3.3, we have $k = k'$ and $K_1(e) = K_2(e')$. The second equation can inductively lead to $K_1 = K_2 \wedge e = e'$, which proves the final conclusion.

□