Iris-OS Documentation

Zhen Zhang

February 28, 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 *Stmts* (statements), *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 $o \in \mathbb{Z}^+$.

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

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

Evaluation Context. To make the evaluation order explicit and reusable by the WP-BIND rule, we define evaluation context. Compared to a simple expression-based language which has only one kind of context K, we define two contexts K_e, K_s for both *Expr* and *Stmts*:

$$
\begin{aligned}
K_e : \textsc{Ectx} &\triangleq \bullet \mid \bullet \oplus e \mid v \oplus \bullet \mid \texttt{!}\bullet \mid \&\bullet \mid \bullet\, \texttt{as}\, \tau \mid \bullet.1 \mid \bullet.2 \\
K_s : \textsc{Sctx} &\triangleq \bullet \leftarrow e \mid l \leftarrow \bullet \mid \texttt{if}(\bullet)\,\{s\}\,\texttt{else}\,\{s\} \mid \texttt{while}(\bullet)\,\{s\} \mid \\
&\qquad \texttt{return}\, \bullet \mid f(v_1, ..., \bullet, e_1, ...)
\end{aligned}
$$

Now we can define *context*: $K : \textsc{Ctx} \triangleq (K_e, K_s)$.

1.2 Semantics

Model. Define $c : \mathit{Code} \triangleq (E_{\text{cur}} : \{\mathit{Expr} + \mathit{Stmts}\}, K_{\text{cur}} : \textsc{Ctx}, K^* : [\textsc{Ctx}])$, in which

- E_{cur} is the "current evaluation", which could be either an expression $e^\dagger$ or a statement $s^\dagger$.
- K_{cur} is the "current context", which semantically means rest code to execute in current frame.
- K^* is the "previous contexts", which semantically means previous frames on call stack

Next, 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 $\sigma : \mathit{State} \triangleq [b \hookrightarrow [v_{\text{ byte}}]]$, i.e. a heap in which every block maps to a list of byte-size values.

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.

Small-Step Operational Semantics. TODO.

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 judgement $\vdash_{\text{tychk}} v : \tau$ 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-INT8-TO-32
$$\frac{\vdash_{\text{tychk}} v : \tau_{\texttt{int8}}}{\vdash_{\text{tychk}} v : \tau_{\texttt{int32}}}$$

TYCHK-INT32-TO-8
$$\frac{\vdash_{\text{tychk}} i \in [0, 2^8) : \tau_{\texttt{int32}}}{\vdash_{\text{tychk}} i : \tau_{\texttt{int8}}}$$

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

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

TYCHK-PTR-CAST
$$\frac{\vdash_{\text{tychk}} v : *\tau}{\vdash_{\text{tychk}} v : *\tau'}$$

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.

Here we first define the type inference rules. Although it is written in an axiomatic way, it is a direct translation of our monadic algorithm (TODO).

Then we define the rules for "interpreting" left-hand side expression (TODO).

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:

$$
\begin{aligned}
b\#o \mapsto_q v : t &\triangleq \exists bs, bs'.\, b \mapsto_q bs * (\text{encode}^{1}(t, v) = bs' \wedge bs' = bs[o..o + |bs'|]) \\
l \mapsto v : t &\triangleq l \mapsto_1 v : t \\
l \mapsto_q - &\triangleq \exists v, t.\, l \mapsto_q v : t
\end{aligned}
$$

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, $S(\sigma) \triangleq \mathsf{Phy}(\bullet\, \mathrm{fmap}(\lambda v.(1, \mathsf{ag} v), \sigma))$.

$$
\begin{aligned}
wp \triangleq \mu\, wp.\, \lambda \mathcal{E}, E_{\mathrm{cur}}, \Phi, \Phi_{\mathrm{ret}}.\ & \\
& (E_{\mathrm{cur}} = \mathtt{skip}^{\dagger} \wedge \Rightarrow_{\mathcal{E}} \Phi(\mathtt{void})) \vee \\
& (\exists v.\, E_{\mathrm{cur}} = v^{\dagger} \wedge \Rightarrow_{\mathcal{E}} \Phi(v)) \vee \\
& (E_{\mathrm{cur}} = \mathtt{return}^{\dagger} \wedge \Rightarrow_{\mathcal{E}} \Phi_{\mathrm{ret}}(\mathtt{void})) \vee \\
& (\exists v.\, E_{\mathrm{cur}} = \mathtt{return}\, v^{\dagger} \wedge \Rightarrow_{\mathcal{E}} \Phi_{\mathrm{ret}}(v)) \vee \\
& \Big(\forall \sigma.\, S(\sigma) \ {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \\
& \qquad \triangleright \forall E'_{\mathrm{cur}}, \sigma'.\, (E_{\mathrm{cur}}, \sigma \to E'_{\mathrm{cur}}, \sigma') \ {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \\
& \qquad\quad S(\sigma') * wp(\mathcal{E}, E'_{\mathrm{cur}}, \Phi, \Phi_{\mathrm{ret}})\Big)
\end{aligned}
$$

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.

[1] see Coq implementation for details

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

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

WP-SKIP
$$\Phi(\mathtt{void}) \vdash \mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi; \Phi_{\mathrm{ret}}\}$$

WP-RET
$$\Phi_{\mathrm{ret}}(v) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{return}\, v\, \{\Phi; \Phi_{\mathrm{ret}}\}$$

WP-BIND
$$\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \mathsf{wp}_{\mathcal{E}}\, K(x)\, \{\Phi; \Phi_{\mathrm{ret}}\}; \Phi_{\mathrm{ret}}\} \vdash \mathsf{wp}_{\mathcal{E}}\, K(e)\, \{\Phi; \Phi_{\mathrm{ret}}\}$$

WP-BIND-E
$$\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \mathsf{wp}_{\mathcal{E}}\, K_e(x)\, \{\Phi; \Phi_{\mathrm{ret}}\}; \Phi_{\mathrm{ret}}\} \vdash \mathsf{wp}_{\mathcal{E}}\, K_e(e)\, \{\Phi; \Phi_{\mathrm{ret}}\}$$

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

WP-ASSIGN
$$\frac{\vdash_{\mathrm{tychk}}\, v : \tau}{l \mapsto - : \tau * (l \mapsto v : \tau \mathrel{-\!\!*} \Phi) \vdash \mathsf{wp}_{\mathcal{E}}\, l \leftarrow v\, \{\Phi; \Phi_{\mathrm{ret}}\}}$$

WP-ASSIGN-OFFSET
$$\frac{\vdash_{\mathrm{tychk}}\, v_2 : \tau_2}{b\#o \mapsto (v_1, -) : \tau_1 \times \tau_2 * (b\#o \mapsto (v_1, v_2) : \tau_1 \times \tau_2 \mathrel{-\!\!*} \Phi) \vdash \mathsf{wp}_{\mathcal{E}}\, b\#(o + \mathrm{sizeof}(\tau_1)) \leftarrow v_2\, \{\Phi; \Phi_{\mathrm{ret}}\}}$$

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

WP-SEQ
$$\mathsf{wp}_{\mathcal{E}}\, s_1\, \{\mathsf{wp}_{\mathcal{E}}\, s_2\, \{\Phi; \Phi_{\mathrm{ret}}\}; \Phi_{\mathrm{ret}}\} \vdash \mathsf{wp}_{\mathcal{E}}\, s_1; s_2\, \{\Phi; \Phi_{\mathrm{ret}}\}$$