Iris$^{\mathcal{C}}$ Documentation

Zhen Zhang

April 13, 2017

Abstract

This document describes the OS verification framework based on the Iris program logic. We omitted many details on Iris itself, which can be found in the document and Coq formalization here https://gitlab.mpi-sws.org/FP/iris-coq/.

Contents

1 Language

1.1 Definitions

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

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

We also have following non-syntactical definitions:

- block address $b \in \mathbb{N}$
- offset $o \in \mathbb{Z}^+$
- Memory address $l : \mathit{Addr} \triangleq b\#o$
- Program $\Sigma : \textsc{Text} \triangleq [x \hookrightarrow \mathit{Function}]$ is a list of functions indexed by names
- Function $F : \mathit{Function} \triangleq (\tau_{ret} \times (x_1 : \tau_1, ...) \times e)$ consists of return type, parameter declarations and function body.

(NOTE) The relationship between $!\, e$ and $!_\tau\, e$: The former one is only used in source, and the later one is used in the actual semantics and inference rules. By adding a pre-processing step which instantiates all source-only expression to substituted/typed ones, we can remove lexical and typing environment from core program logic. The details are in 1.3

(NOTE) Compared with C, there are several notable differences:

1. We currently don't support declarations of local variables, `for` loop and `switch`, `break` and `continue`
2. C differentiates statements and expressions (so do many other program logic). For simplicity and expressivity, we merged them together into a coherent expression *Expr* instead

Evaluation Context. To make the evaluation order explicit and reflected into the logic through the bind rule, we define evaluation context:

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

1.2 Semantics

Model. First define

- byte-size value
$$v_{\text{byte}} : \mathit{Val}_{\text{byte}} \triangleq \lightning \mid i \in [0, 2^8) \mid l_{\{0|1|2|3\}} \mid \texttt{null}$$
- Heap $h : \textsc{Heap} \triangleq [l \hookrightarrow v_{\text{byte}}]$
- Continuation $K : \textsc{Cont} \triangleq [\textsc{Ectx}]$
- Stack $s : \textsc{Stack} \triangleq [\textsc{Cont}]$
- Whole state
$$\sigma : \mathit{State} \triangleq (\text{heap} : \textsc{Heap}, \text{text} : \textsc{Text}, \text{stack} : \textsc{Stack})$$

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

Small-Step Operational Semantics. We define the small-step semantics (see figure 1.2) for both local reduction ($(e, h) \to_{local} (e', h')$) and non-local reductions ($(e, s : \textsc{Stack}) \to^{\Sigma}_{jump} (e', s' : \textsc{Stack})$), and then combine them together ($(e, \sigma) \to_c (e', \sigma')$) point-wise.

ES-BINOP
$$\frac{[\![oplus]\!](v_1, v_2) = v'}{(v_1 \oplus v_2, h) \to_{local} (v', h)}$$

ES-DEREF-TYPED
$$\frac{\vdash_{\text{typeof}} v : \tau \quad h \vdash l \mapsto \text{encode}(v)}{(\,!_\tau\, l, h) \to_{local} (v, h)}$$

ES-FST
$$((v_1, v_2).1, h) \to_{local} (v_1, h)$$

ES-SND
$$((v_1, v_2).2, h) \to_{local} (v_2, h)$$

ES-ASSIGN
$$(l \leftarrow v, h) \to_{local} (\texttt{skip}, h[l \mapsto \text{encode}(v)])$$

ES-SEQ
$$(v; e, h) \to_{local} (e, h)$$

ES-ALLOC
$$\frac{\vdash_{\text{typeof}} v : \tau \quad \forall o'.\, h(b, o') = \bot}{(\texttt{alloc}_\tau(v), h) \to_{local} (b\#o, h[b\#o \mapsto v])}$$

ES-WHILE-TRUE
$$(\texttt{while}_c(\mathsf{true})\,\{s\}, h) \to_{local} (s; \texttt{while}_c(c)\,\{s\}, h)$$

ES-WHILE-FALSE
$$(\texttt{while}_c(\mathsf{false})\,\{s\}, h) \to_{local} (\texttt{skip}, h)$$

ES-BIND'
$$\frac{\text{is\_jump}(e) = \mathsf{False} \quad (e, h) \to_{local} (e', h')}{((k :: ks)e, h) \to_{local} ((k :: ks)e', h')}$$

JS-RETE
$$\frac{\text{unfill}(k(\texttt{return}\, v)) = (k', \texttt{return}\, v)}{(k'(\texttt{return}\, v), k :: ks) \to^{\Sigma}_{jump} (k(v), ks)}$$

JS-CALL
$$\frac{\Sigma(f) = \mathit{Function}(\_, ps, e)}{(k(f(ls)), ks) \to^{\Sigma}_{jump} (e[ps/ls], k :: ks)}$$

Figure 1: Semantics rules

1.3 Type System and Lexical 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 $\vdash_{\text{typeof}} v : \tau$ for values.

$$\text{TYPEOF-VOID}\quad \vdash_{\text{typeof}} \texttt{void} : \tau_{\texttt{void}} \qquad \text{TYPEOF-NULL}\quad \vdash_{\text{typeof}} \texttt{null} : \tau_{\texttt{null}} \qquad \text{TYPEOF-INT8}\quad \vdash_{\text{typeof}} i \in [0, 2^8) : \tau_{\texttt{int8}} \qquad \text{TYPEOF-INT32}\quad \vdash_{\text{typeof}} i \in [0, 2^{32}) : \tau_{\texttt{int32}}$$

$$\text{TYPEOF-NULL-PTR}\quad \forall \tau.\ \vdash_{\text{typeof}} \texttt{null} : *\tau \qquad \text{TYPEOF-PTR}\quad \forall \tau, l.\ \vdash_{\text{typeof}} l : *\tau \qquad \text{TYPEOF-PROD}\quad \frac{\vdash_{\text{typeof}} v_1 : \tau_1 \quad \vdash_{\text{typeof}} v_2 : \tau_2}{\vdash_{\text{typeof}} (v_1, v_2) : \tau_1 \times \tau_2}$$

Note that rule TYPEOF-NULL-PTR means that `null` can be of any pointer type, and TYPEOF-PTR means that a pointer can have *any* pointer type.

Lexical Environment In Iris$^{\mathcal{C}}$, before we do any real work over a function body, variables are all replaced with their bindings, and type information is tagged to the untyped parts when in need. More specifically, we will replace variables with either their location (when in left-hand side) or the dereference of their location (when in right-hand side). We will produce a pointer arithmetic expression when processing the left-hand side expression, which in turn requires type inference $\vdash_{\text{tyinf}} e : \tau$ (since it is standard and trivial, we will leave out the details here).

Now we write down the algorithmic rules for rewriting left-hand side expression ($(\!|e|\!)_{\text{LHS}}$) and right-hand side expression ($(\!|e|\!)_{\text{RHS}}$). Essentially, $(\!|e|\!)_{\text{LHS}}$ will rewrite the left-hand side e into its location in memory, while $(\!|e|\!)_{\text{RHS}}$ will rewrite variables into the dereference of its locations.

Note that environment $\Gamma : \mathit{Env} \triangleq [x \hookrightarrow (\tau, l)]$ is assumed, and if any operation fails, like missing something in Γ, then the rule will indicate it in the actual implementation. And also, the cases not covered are defined trivially.

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

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

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:

$$
\begin{aligned}
l \mapsto_q v : t &\triangleq l \mapsto_q \mathrm{encode}(v) \land \vdash_{\mathrm{typeof}} v : t \\
l \mapsto v : t &\triangleq l \mapsto_1 v : t \\
l \mapsto_q - : t &\triangleq \exists v.\, l \mapsto_q v : t \\
\mathrm{own\_stack}(s) &\triangleq \boxed{(\tfrac{1}{2}, \mathsf{ag} s)}^{\textsc{Stack}} \\
f \mapsto_{\textsc{Text}} F &\triangleq \boxed{\circ\, [f \leftarrow \mathsf{ag} F]}^{\textsc{Text}}
\end{aligned}
$$

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 can be derived.

Defining weakest precondition. The state interpreting predicate $S : \mathit{State} \to \mathit{iProp}$ for our Iris$^{\mathcal{C}}$ language is defined below, as required for instantiating the parametric WP of Iris (whose definition is also copied below for reference).

$$
S(\sigma) \triangleq \boxed{\bullet\, \mathsf{fmap}(\lambda v.(1, \mathsf{ag} v), \sigma.\mathrm{heap})}^{\textsc{Heap}} * \boxed{\bullet\, \mathsf{fmap}(\mathsf{ag}, \sigma.\mathrm{text})}^{\textsc{Text}} * \boxed{(\tfrac{1}{2}, \sigma.\mathrm{stack})}^{\textsc{Stack}}
$$

$$
\begin{aligned}
wp \triangleq\ & \mu\, wp.\, \lambda \mathcal{E}, e, \Phi. \\
& (\exists v.\, \mathrm{to\_val}(e) = v \land \Rrightarrow_{\mathcal{E}} \Phi(v)) \lor \\
& \Big(\mathrm{to\_val}(e) = \bot \land \\
& \quad \forall \sigma.\, S(\sigma) \stackrel{\mathcal{E}}{\Longrightarrow\!\!\!\!*}{}^{\emptyset} \\
& \quad \mathrm{red}(e, \sigma) * \triangleright \forall e', \sigma'.\, (e, \sigma) \to_c (e', \sigma') \stackrel{\emptyset}{\Longrightarrow\!\!\!\!*}{}^{\mathcal{E}} \\
& \qquad S(\sigma') * wp(\mathcal{E}, e', \Phi)\Big)
\end{aligned}
$$

Here are some conventions:

- If we leave away the mask $\mathcal{E}$, we assume it to be $\top$.
- 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-STRONG-MONO
$$\frac{\mathcal{E}_1 \subseteq \mathcal{E}_2}{((\forall v.\, \Phi(v) \Rrightarrow_{\mathcal{E}_2} \Psi(v)) \land (\forall v.\, \Phi_{\mathrm{ret}}(v) \Rrightarrow_{\mathcal{E}_2} \Psi_{\mathrm{ret}}(v))) * \mathsf{wp}_{\mathcal{E}_1}\, E_{\mathrm{cur}}\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}_2}\, E_{\mathrm{cur}}\, \{\Psi\} \Psi_{\mathrm{ret}}}$$

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

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

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

WP-RET
$$\mathrm{own\_stack}(k' :: ks) * (\mathrm{own\_stack}(ks) \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, k'(v)\, \{\Phi\}) \vdash \mathsf{wp}_{\mathcal{E}}\, k(\mathtt{return}\, v)\, \{\Phi\}$$

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

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

WP-ASSIGN
$$\frac{\vdash_{\mathrm{typeof}} v : \tau' \quad \mathrm{assign\_compatible}(\tau \leftarrow \tau')}{\triangleright\, l \mapsto - : \tau * \triangleright (l \mapsto v : \tau \mathrel{-\!\!*} \Phi) \vdash \mathsf{wp}_{\mathcal{E}}\, l \leftarrow v\, \{\Phi\}}$$

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

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

WP-WHILE-TRUE
$$\triangleright\, \mathsf{wp}_{\mathcal{E}}\, s; \mathtt{while}_c(c)\, \{s\}\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{while}_c(\mathsf{true})\, \{s\}\, \{\Phi\}$$

WP-WHILE-FALSE
$$\triangleright\, \Phi(\mathtt{void}) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{while}_c(\mathsf{false})\, \{s\}\, \{\Phi\}$$

WP-WHILE-INV
$$\frac{\mathrm{is\_jump}(s) = \mathsf{False} \qquad \mathrm{is\_jump}(c) = \mathsf{False}}{\begin{array}{c}\forall \Phi.\, (I * (\forall v.\, (v = \mathsf{false} * Q(\mathtt{void})) \lor (v = \mathsf{true} * I)) \mathrel{-\!\!*} \Phi(v)) \mathrel{-\!\!*} \mathsf{wp}\, c\, \{\Phi\} \\ \forall \Phi.\, (I * (I \mathrel{-\!\!*} \Phi(\mathtt{void}))) \mathrel{-\!\!*} \mathsf{wp}\, s\, \{\Phi\}\end{array}}{I \vdash \mathsf{wp}\, \mathtt{while}_c(c)\, \{s\}\, \{Q\}}$$

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
$$\frac{\vdash_{\mathrm{typeof}} v : \tau}{(\forall l.\, l \mapsto v : \tau \mathrel{-\!\!*} \Phi(l)) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{alloc}_\tau(v)\, \{\Phi\}}$$

WP-CALL
$$f \mapsto_{\mathrm{TEXT}} \mathit{Function}(\_, ps, e) * \mathrm{own\_stack}(ks) * \triangleright (\mathrm{own\_stack}(k :: ks) \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e[ps/ls]\, \{\Phi\}) \vdash \mathsf{wp}_{\mathcal{E}}\, k(f(ls))\, \{\Phi\}$$

2.3 Extending Weakest Precondition

We extend the unary-exit WP into a binary-exit $\mathsf{wp}^+_{\mathcal{E}}\, e\,\{\Phi\,;\,\Phi_{\mathrm{ret}}\}$, by defining it as below:

$$
\begin{aligned}
wp^+ \triangleq\ & \mu\, wp^+.\, \lambda \mathcal{E}, e, \Phi, \Phi_{\mathrm{ret}}. \\
& (\exists v.\, \mathrm{to\_val}(e) = v \wedge \Phi(v)) \vee \\
& (\mathrm{to\_val}(e) = \bot \wedge \exists e_h, K.\, e = K(e_h) \wedge \Big(\\
& \quad (\mathrm{is\_jump}(e_h) = \mathsf{False} * \mathsf{wp}_{\mathcal{E}}\, e_h \left\{ v.\, \mathsf{wp}^+_{\mathcal{E}}\, K(v)\,\{\Phi\,;\,\Phi_{\mathrm{ret}}\} \right\}) \vee \\
& \quad (\exists v.\, e_h = \mathtt{return}\, v * \triangleright \Phi_{\mathrm{ret}}(v)) \vee \\
& \quad (\exists f, ps, ls, F. \\
& \qquad f \mapsto_{\mathrm{TEXT}} \mathit{Function}(\_, ps, F) * \\
& \qquad \triangleright(\mathsf{wp}^+_{\mathcal{E}}\, F[ps/ls] \left\{ \_.\, \mathsf{False}\,;\, v.\, \mathsf{wp}^+_{\mathcal{E}}\, K(v)\,\{\Phi\,;\,\Phi_{\mathrm{ret}}\} \right\})) \\
& \Big))
\end{aligned}
$$

And this definition supports the following inference rules:

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

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

WPR-BIND
$$\mathsf{wp}^+_{\mathcal{E}}\, e \left\{ x.\, \mathsf{wp}^+_{\mathcal{E}}\, k(x)\,\{\Phi\,;\,\Phi_{\mathrm{ret}}\}\,;\,\Phi_{\mathrm{ret}} \right\} \vdash \mathsf{wp}^+_{\mathcal{E}}\, k(e)\,\{\Phi\,;\,\Phi_{\mathrm{ret}}\}$$

WPR-CALL
$$f \mapsto_{\mathrm{TEXT}} \mathit{Function}(\_, ps, e) * \triangleright(\mathsf{wp}^+_{\mathcal{E}}\, e[ps/ls]\,\{\_.\, \mathsf{False}\,;\,\Phi\}) \vdash \mathsf{wp}^+_{\mathcal{E}}\, k(f(ls))\,\{\Phi\,;\,\Phi_{\mathrm{ret}}\}$$

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

Note how WP-CALL, WP-RET, and WP-BIND are simplified in their new, corresponding versions, also the fact that we can recover local evaluation like WP-OP trivially (and we won't duplicate too much here).

2.4 Soundness

The soundness of WP-style program is proven by showing that it is *adequate*: For all $e, \sigma, \Phi : \mathit{Val} \to \mathsf{Prop}$,

$$
\begin{aligned}
& \mathsf{True} \vdash \mathsf{wp}_{\top}\, e\,\{\Phi\} \to \\
& \quad (\forall v, \sigma'.\, (e, \sigma) \to_c (v, \sigma') \to \Phi(v)) \wedge \\
& \quad (\forall e', \sigma'.\, (e, \sigma) \to_c (e', \sigma') \to (\exists v.\, e = v) \vee \mathrm{red}(e', \sigma'))
\end{aligned}
$$

(NOTE) However, that doesn't look strong enough.

3 Automation

We also developed some basic tactics for automatically solving the goals, mostly related to our new language, some enhancing what Iris provides.

3.1 Tactics for "Symbolic Execution"

Similar to what exists in Iris's heap-lang, we also provide convenient tactics including:

- `wp_bind <p>`: bind to a head term, or a term containing the head term, with pattern `p`
- `wp_assign`: make a step by evaluating assignment's head term $l \leftarrow v$ (it also shifts following statements by applying `wp_seq` repetitively beforehand)
- `wp_load`, `wp_op` are similar to the one above
- `wp_skip`: skip over any value before a sequencing operator
- `wp_run`: keep executing as long as any of the above applies, which feels like doing symbolic execution automatically

3.2 Other Tactics

- `gmap_simplify`: simplify expressions involving `gmap` based on some algebraic rules

4 Refinement

4.1 Spec State and Spec Code

Considering a STS composed of spec code $\tilde{c}$ (as below), spec state $\sigma : [X \hookrightarrow v]$, and semantic rules $(\tilde{c}, \tilde{\sigma}) \rightarrow_{spec} (\tilde{c}', \tilde{\sigma}')$ (as well as $(\tilde{c}, \tilde{\sigma}) \rightarrow^*_{spec} (\tilde{c}', \tilde{\sigma}')$ trivially derived).

$$\tilde{c} : \textsc{SpecCode} ::= \texttt{done}(v?) \mid \texttt{rel}(r : \tilde{\sigma} \to v? \to \tilde{\sigma} \to \mathsf{Prop}) \mid ...$$

4.2 Refinement RA

We defined a new RA Refine to capture the history of spec code execution:

$$\textsc{View} \triangleq \mathsf{master} \mid \mathsf{snapshot}$$
$$\textsc{Refine} \triangleq \textsc{View} \times [\textsc{SpecState} \times \textsc{SpecCode}]$$

The validity of Refine:

$$\checkmark(\mathsf{refine}(v, \emptyset)) \qquad \checkmark(\mathsf{refine}(v, [(\tilde{\sigma}, \tilde{c})])) \qquad \frac{(\tilde{c}, \tilde{\sigma}) \rightarrow_{spec} (\tilde{c}', \tilde{\sigma}') \qquad \checkmark(\mathsf{refine}(v, [(\tilde{\sigma}, \tilde{c}) :: cs]))}{\checkmark(\mathsf{refine}(v, [(\tilde{\sigma}', \tilde{c}') :: (\tilde{\sigma}, \tilde{c}) :: cs]))}$$

The multiplication of View and Refine is defined as:

$$\mathsf{master} \cdot \_ \triangleq \mathsf{master}$$
$$\_ \cdot \mathsf{master} \triangleq \mathsf{master}$$
$$\mathsf{snapshot} \cdot \mathsf{snapshot} \triangleq \mathsf{snapshot}$$

$$\mathsf{refine}(v_1, cs_1) \cdot \mathsf{refine}(v_2, cs_2) \triangleq \begin{cases} \mathsf{refine}(v_1 \cdot v_2, cs_1) & |cs_1| \geq |cs_2| \\ \mathsf{refine}(v_1 \cdot v_2, cs_2) & |cs_1| < |cs_2| \end{cases}$$

The disjointness of multiplication is refined as:

$$\frac{\exists cs'.\, cs_1 +\!\!+ cs = cs_2 \vee \exists cs'.\, cs_2 +\!\!+ cs = cs_1}{\mathsf{refine}(\mathsf{snapshot}, cs_1) \# \mathsf{refine}(\mathsf{snapshot}, cs_2)} \qquad \mathsf{refine}(\mathsf{snapshot}, cs_1) \# \mathsf{refine}(\mathsf{master}, cs_1 +\!\!+ cs_1)$$

$$\mathsf{refine}(\mathsf{master}, cs_1 +\!\!+ cs_1) \# \mathsf{refine}(\mathsf{snapshot}, cs_1)$$

In the end, Refine can be proven to be a CMRA with an unit element $\mathsf{refine}(\mathsf{snapshot}, \emptyset)$.

4.3 Refinement Ghost State, Invariant and Rules

The refinement ghost state `refineG` contains three part: one for $refineM$, and two for *paired* ownership of SpecCode and SpecState. The predicates are defined as below:

$$\begin{aligned}
\text{sstate}(\tilde{\sigma}) &\triangleq \boxed{(\tfrac{1}{2}, \mathsf{ag}\tilde{\sigma})}^{\textsc{SpecState}} \\
\text{scode}(\tilde{c}) &\triangleq \boxed{(\tfrac{1}{2}, \mathsf{ag}\tilde{c})}^{\textsc{SpecCode}} \\
\text{master}'(cs) &\triangleq \boxed{\mathsf{refine}(\mathsf{master}, cs)}^{\textsc{Refine}} \\
\text{master}(c) &\triangleq \exists cs. \boxed{\mathsf{refine}(\mathsf{master}, c :: cs)}^{\textsc{Refine}} \\
\text{snapshot}'(cs) &\triangleq \boxed{\mathsf{refine}(\mathsf{snapshot}, cs)}^{\textsc{Refine}} \\
\text{snapshot}(c) &\triangleq \exists cs. \boxed{\mathsf{refine}(\mathsf{snapshot}, c :: cs)}^{\textsc{Refine}}
\end{aligned}$$

Then we define refinement invariant:

$$I_{\textsc{Refine}} \triangleq \exists \tilde{\sigma}, \tilde{c}.\, \text{sstate}(\tilde{\sigma}) * \text{scode}(\tilde{c}) * \text{master}(\tilde{\sigma}, \tilde{c})$$

With this, we can give refinement style proofs for certain kernel APIs, using the following derived rules:

SPEC-UPDATE

$$\frac{(\tilde{c}, \tilde{\sigma}) \rightarrow_{spec} (\tilde{c}', \tilde{\sigma}')}{\boxed{I_{\textsc{Refine}}}^{\iota} * \text{sstate}(\tilde{\sigma}) * \text{scode}(\tilde{c}) \vdash \Rrightarrow \boxed{I_{\textsc{Refine}}}^{\iota} * \text{sstate}(\tilde{\sigma}') * \text{scode}(\tilde{c}') * \text{snapshot}(\tilde{\sigma}', \tilde{c}')}$$

5 Misc

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

5.1 Basic 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 occurrences.

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

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

Theorem 5.1. *For any $e, e' : \mathit{Expr}$ and $k_{im}, k_{jn} : \text{Ectx}$,*

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

Proof. Trivial. □

Theorem 5.2. *For any $e, e' : \mathit{Expr}$ and $k_{im}, k_{in} : \text{Ectx}_i$,*

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

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. □

As you may observe, the expression space spanned by $k(e)$ for any k, e is not the entire expression space as an ADT. Instead, it is a subset of *Expr* which represents well-formed ones that can appear as an immediate form according to some well-defined evaluation order. We call such e is *well-formed*.

5.2 Axioms about Evaluation Context based Semantics

In the following lemmas, we assume all involved e to be *well-formed*.

Lemma 5.3. $\forall e : \mathit{Expr}, k : \textsc{Cont}.\, \text{is\_enf}(e) \to \text{unfill}(k(e)) = (k, e)$

Proof. Let's prove inductively w.r.t k. When k is empty, since e is in normal form, so unfill it will only return the same thing. And inductively, since fill and unfill should cancel out, the final conclusion is proved trivially as well.

□

Lemma 5.4. $\forall e, e_h, k.\, \text{unfill}(e) = (k, e_h) \to \text{is\_enf}(e_h) \wedge e = k(e_h)$

Proof. First, e can't be a value, so we just need to consider e in each legal case, which basically mean that either e is normal form itself, or $e = K(e')$ and $\exists k', k = K :: k' \wedge \text{unfill}(e') = (k', e_h)$.

1. In the first case, k is forced to be $\emptyset$ and $e_h = e$, so this case is proved
2. In the second case, we can inductively know that e_h is normal form and $e' = k'(e_h)$, so $e = K(k'(e_h)) = (K :: k')(e_h) = k(e_h)$, so this case is proved as well

□

Lemma 5.5. *Induction scheme I for Expr:*

$$\begin{aligned}\forall P : \mathit{Expr} \to \mathit{Prop}.\, &(\forall e.\, \text{is\_enf}(e) \to P(e)) \to \\ &(\forall e, k : \textsc{Cont}.\, \text{to\_val}(e) = \bot \to P(e) \to P(k(e))) \to \\ &(\forall e.\, \text{to\_val}(e) = \bot \to P(e))\end{aligned}$$

Proof. We want to *inductively* prove that for any non-value, *well-formed* e, $P(e)$ holds. Note that P here can be any proposition.

The base case is when e is in normal form, which corresponds to the first condition; The inductive case is that for any well-formed e' that is not in normal form, it must be of the form $k(e)$ for some k, e. With each proved given sufficient inductive assumption, we know that any well-formed e must satisfy P. □

Lemma 5.6. *Induction scheme for* Cont*:*

$$\begin{aligned}\forall P : \textsc{Cont} \to \mathit{Prop}.\, &(P(\emptyset)) \to \\ &(\forall k.\, (\forall.\, k', |k'| < |k| \to P(k')) \to P(k)) \to \\ &(\forall k.\, P(k))\end{aligned}$$

Proof. Intuitively, this is about property of natural number as the length of k. When we know $P(\emptyset)$, we know $\forall K_1.\, P(K_1 :: [])$, then we know $\forall K_2.\, \forall K_1.\, P(K_2 :: K_1 :: [])$ .... Until for arbitrarily big, finite n, $\forall K_n, K_{n-1}, ..., K_1.P(K_n :: K_{n-1} :: ... :: K_1 :: [])$, or for any finite k, $P(k)$ holds. □

Lemma 5.7. *Induction scheme II for Expr:*

$$\begin{aligned}\forall P : \mathit{Expr} \to \mathit{Prop}.\, &(\forall e.\, \text{is\_enf}(e) \to P(e)) \to \\ &(\forall e, k : \textsc{Cont}.\, \text{is\_enf}(e) \to (\forall k'.\, |k'| < |k| \to P(k'(e))) \to P(k(e))) \to \\ &(\forall e.\, \text{to\_val}(e) = \bot \to P(e))\end{aligned}$$

Proof. First, the e in the final goal is implicitly well-formed, which means that $e = K(e_h)$ for some K, e_h. So we prove inductively on the length of such K, in a similar way to the last lemma, and reach our final conclusion. □

Lemma 5.8. *Partial injectivity of fill:*

$$\frac{\text{to\_val}(e) = \bot \quad \text{is\_enf}(e_h) \quad k(e) = k'(e_h)}{\exists k''.\, k' = k +\!\!+\, k'' \qquad e = k''(e_h)}$$

Proof. Since e is well-formed and not a value, so it can be unfilled such that $e = k''(e'_h)$, then by `cont_inj`, $e_h = e'_h$ and $k' = k + +k''$, which proves our claim

□

Lemma 5.9. *Local step has a focus:*

$$\frac{(e_1, h_1) \to_{local} (e_2, h_2)}{\exists e'_1, e'_2, k.\, \text{is\_enf}(e'_1) \wedge e_1 = k(e'_1) \wedge e_2 = k(e'_2) \wedge (e'_1, h_1) \to_{local} (e'_2, h_2)}$$

Proof. By inversing the assumption, we know there are two possibilities:

1. If e_1 is already in normal form, then we just need to let $e'_1 = e_1, e'_2 = e_2$, and $k = \emptyset$.
2. We know that there are some k'', e''_1, e''_2, where $|k''| > 0$, such that $e_1 = k''(e''_1), e_2 = k''(e''_2)$, and $(e''_1, h_1) \to_{local} (e''_2, h_2)$. Since by the metric of depth $||_d$, $|e''_1|_d < |e_1|_d$, so we can derive inductively that there exists e'_1, e'_2, k' that

   $$\text{is\_enf}(e'_1) \wedge e''_1 = k'(e'_1) \wedge e''_2 = k'(e'_2) \wedge (e'_1, h_1) \to_{local} (e'_2, h_2)$$

   Now let $k = k'' +\!\!+\, k'$, and use the same e'_1, e'_2, we can prove the original existential goal.

□

5.3 Other Facts

Lemma 5.10. *wp^+ is contractive.*

Proof. Observe in the definition 2.3 that there are only two recursive uses of wp^+:

1. In the second branch of non-value case, we use wp^+ under Φ of $\mathsf{wp}_{\mathcal{E}}\, e_h\, \{\Phi\}$, which means that,

   $$\mathsf{wp}_{\mathcal{E}}\, e_h \left\{ ... \mathsf{wp}^+ ... \right\} \overset{n}{=} \mathsf{wp}_{\mathcal{E}}\, e_h \left\{ ... \mathsf{wp}'^+ ... \right\}$$

   will hold if $\mathsf{wp}^+ \overset{n+1}{=} \mathsf{wp}'^+$, which is satisfied by assumption.
2. In the third branch of non-value case, we use wp^+ under a $\triangleright$, which is trivial to prove by `f_contractive`.

□

5.4 Admitted Facts

There are still one or more loopholes that exists behind the documented formalization. But note that all of them are harmless to leave unproven for now.

- In `lang.v`, lemma `same_type_encode_inj`'s fourth case on `int32` is admitted. Essential, this is about encoding `int32` value with four bytes. It is trivially true, and even if it is false, the soundness of logic won't suffer. We don't prove it now because there are not enough lemmas exposed from `Integers.v`.