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

Zhen Zhang

September 12, 2017

Abstract

This document describes the control-flow based language verification framework based on the Iris program logic. The Coq source is in https://github.com/izgzhen/iris-c-coq. 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 *Expr* (expressions), *Type* (types) and *Val* (values).

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

We also have following definitions:

- block address $b \in \mathbb{N}$
- offset $o \in \mathbb{Z}^+$
- Memory address $l : \mathit{Addr} \triangleq b\#o$
- Program $\Gamma : \textsc{Text} \triangleq [f \hookrightarrow \mathit{Function}]$ 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) Compared with C, there are several notable differences:

1. We currently don't support `for` loop and `switch`
2. C differentiates statements and expressions (so do many other program logics). For simplicity and expressivity, we merged them together into a coherent expression *Expr* instead
3. We support declarations of local variables through a syntactic sugar based on the core language (definition of default : *Type* → *Val* is trivial)

$$\texttt{let}\, x : \tau\, \texttt{in}\, e \triangleq \texttt{let}\, x : \tau := \texttt{alloc}_\tau(\mathrm{default}(\tau))\, \texttt{in}\, e$$

(NOTE) We sometimes use `c` : *Expr* to denote *condition* expression in `if` and `while`.

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

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

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 $h : \textsc{Heap} \triangleq [l \hookrightarrow v_{\text{ byte}}]$

- Text $\Gamma : \textsc{Text} \triangleq [f \hookrightarrow \mathit{Function}]$

- Continuation $k : \textsc{Cont} \triangleq [\textsc{Ectx}]$

- Context $K : \textsc{Ctx} \triangleq K_{while}(c, e, k) \mid K_{call}(k, \epsilon)$

- Stack $s : \textsc{Stack} \triangleq [\textsc{Ctx}]$

- Environment: $\epsilon : [x \hookrightarrow (\tau, v)]$

- Shared state
$$\sigma : \mathit{State} \triangleq (\text{heap} : \textsc{Heap}, \text{text} : \textsc{Text})$$

- Local state
$$\sigma_l : \mathit{LocalState} \triangleq (\text{stack} : \textsc{Stack}, \text{env} : \mathit{Env})$$

(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 three sub-semantics:

- local reduction $(\epsilon, e, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon', e', h', (t_1, t_2, ...))$ (when the set of forked-off threads $t_1, ...$ is empty, we abbreviate it as $(\epsilon, e, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon', e', h')$)

- *jump* reductions $(\epsilon, e, s) \rightsquigarrow^{\Gamma}_{jump} (\epsilon', e', s')$

- *while* reductions $(e, s) \rightsquigarrow_{while} (e', s')$

Then we combine them together into $(\Gamma, \epsilon, e) \rightsquigarrow (s, \sigma, \epsilon, e', (s', \sigma'))t_1, ...$ point-wise.

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 the 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.

TYPEOF-VOID
$\vdash_{\text{typeof}} \texttt{void} : \tau_{\texttt{void}}$

TYPEOF-NULL
$\vdash_{\text{typeof}} \texttt{null} : \tau_{\texttt{null}}$

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

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

TYPEOF-NULL-PTR
$\forall \tau.\ \vdash_{\text{typeof}} \texttt{null} : \text{Ptr}(\tau)$

TYPEOF-PTR
$\forall \tau, l.\ \vdash_{\text{typeof}} l : \text{Ptr}(\tau)$

TYPEOF-PROD
$$\frac{\vdash_{\text{typeof}} v_1 : \tau_1 \qquad \vdash_{\text{typeof}} v_2 : \tau_2}{\vdash_{\text{typeof}} (v_1, v_2) : \tau_1 \times \tau_2}$$

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 Formally, the lexical environment for a function body consists of two parts: parameter bindings (free variables) and local bindings (introduced by `let`).

We model the C semantics that these two types of bindings are essentially the same, except that we initialize the parameter bindings by passed in nested value pair (note that it is in the value domain, rather than a Coq-level list, which simplifies the verification), and initialize the local bindings by placeholder values.

ES-BINOP
$$\frac{[\![oplus]\!](v_1, v_2) = v'}{(\epsilon, v_1 \oplus v_2, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon, v', h)}$$

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

ES-FST
$$(\epsilon, (v_1, v_2).1, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon, v_1, h)$$

ES-SND
$$(\epsilon, (v_1, v_2).2, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon, v_2, h)$$

ES-ASSIGN
$$(\epsilon, l \leftarrow v, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon, \texttt{void}, h[l \mapsto \text{encode}(v)])$$

ES-SEQ
$$(\epsilon, v; ; e, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon, e, h)$$

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

ES-IF-TRUE
$$(\epsilon, \texttt{if}(\mathsf{true})\, \{e_1\}\, \texttt{else}\, \{e_2\}, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon, e_1, h)$$

ES-IF-FALSE
$$(\epsilon, \texttt{if}(\mathsf{false})\, \{e_1\}\, \texttt{else}\, \{e_2\}, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon, e_2, h)$$

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

ES-CAS-FAIL
$$\frac{\vdash_{\text{typeof}} v_1 : \tau \quad \vdash_{\text{typeof}} v_2 : \tau \quad \vdash_{\text{typeof}} v : \tau \quad h \vdash l \mapsto \text{encode}(v) \quad v_1 \neq v}{(\epsilon, \texttt{CAS}(l, v_1, v_2) : \tau, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon, \mathsf{false}, h)}$$

ES-CAS-SUC
$$\frac{\vdash_{\text{typeof}} v_1 : \tau \quad \vdash_{\text{typeof}} v_2 : \tau \quad h \vdash l \mapsto \text{encode}(v_1)}{(\epsilon, \texttt{CAS}(l, v_1, v_2) : \tau, h) \rightsquigarrow^{\Gamma}_{local} (\epsilon, \mathsf{true}, h[l \mapsto \text{encode}(v_2)])}$$

ES-FORK
$$\frac{\Gamma(f) = \mathit{Function}(\tau, [], e)}{(\epsilon, \texttt{fork}(f() : \tau), h) \rightsquigarrow^{\Gamma}_{local} (\epsilon, \texttt{void}, h, (\{e[ps/vs]\}))}$$

JS-RETE
$$\frac{)}{(\epsilon, k'(\texttt{return}\, v), KS' +\!\!+\, K_{call}(k, :) : KS) \rightsquigarrow^{\Gamma}_{jump} (\epsilon, k(v), KS)}$$

JS-CALL
$$\frac{\Sigma(f) = \mathit{Function}(\tau, ps, e)}{(\epsilon, k(f(vs) : \tau), KS) \rightsquigarrow^{\Gamma}_{jump} (\epsilon, e[ps/vs], K_{call}(k, :) : KS)}$$

WS-WHILE
$$(k(\texttt{while}(c)\, \{e\}), KS) \rightsquigarrow_{while} (\texttt{if}(c)\, \{e; \texttt{continue}\}\, \texttt{else}\, \{\texttt{break}\}, K_{while}(c, e, k) :: KS)$$

WS-BREAK
$$\frac{\forall K \in KS'.\, K \neq K_{while}(\_, \_, \_)}{(k'(\texttt{break}), KS' +\!\!+\, K_{while}(c, e, k) :: KS) \rightsquigarrow_{while} (k(\texttt{void}), KS)}$$

WS-CONTINUE
$$\frac{\forall K \in KS'.\, K \neq K_{while}(\_, \_, \_)}{(k'(\texttt{continue}), KS' +\!\!+\, K_{while}(c, e, k) :: KS) \rightsquigarrow_{while} (k(\texttt{while}(c)\, \{e\}), KS)}$$

Figure 1: Semantics rules

2 Program Logic

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

2.1 Ghost Resource

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

$$\begin{aligned}
l \mapsto_q v : t &\triangleq l \mapsto_q \text{encode}(v) \land \vdash_{\text{typeof}} v : t \\
l \mapsto v : t &\triangleq l \mapsto_1 v : t \\
l \mapsto_q - : t &\triangleq \exists v.\, l \mapsto_q v : t \\
f \mapsto_{\text{TEXT}} F &\triangleq \boxed{\circ\, [f \leftarrow \mathsf{ag} F]}^{\text{TEXT}}
\end{aligned}$$

2.2 Weakest Precondition

Defining weakest precondition. The shared state interpreting predicate $S : \mathit{State} \to \mathit{iProp}$ for our Iris$^{\mathcal{C}}$ language is defined below, with threads local state s set to STACK $\times$ *Env* (and initialized to $(\emptyset, \emptyset)$), as required for instantiating the parametric WP defined for a general class of `language`.

$$S(\sigma) \triangleq \boxed{\bullet\, \text{fmap}(\lambda v.(1, \mathsf{ag} v), \sigma.\text{heap})}^{\text{HEAP}} * \boxed{\bullet\, \text{fmap}(\mathsf{ag}, \sigma.\text{text})}^{\text{TEXT}}$$

$$\begin{aligned}
\mathit{wp} \triangleq\ & \mu\, \mathit{wp}.\, \lambda \mathcal{E}, (e, s), \Phi. \\
& \quad (\exists v.\, \text{to\_val}(e) = v \land \Rrightarrow_{\mathcal{E}} \Phi(v)) \lor \\
& \quad \Big(\text{to\_val}(e) = \bot \land \\
& \qquad \forall \sigma.\, S(\sigma)\ {}^{\mathcal{E}}\!\Rrightarrow\!\!\ast^{\emptyset} \\
& \qquad \text{red}(e, s, \sigma) * \triangleright \forall e', s', \sigma', \vec{e}.\, (e, s, \sigma) \leadsto (e', s', \sigma', (\vec{e}))\ {}^{\emptyset}\!\Rrightarrow\!\!\ast^{\mathcal{E}} \\
& \qquad\quad S(\sigma') * \mathit{wp}(\mathcal{E}, (e', s'), \Phi) * \mathop{\Large\ast}_{e'' \in \vec{e}} \mathit{wp}(\top, (e'', \text{init}_s), \lambda\_.\, \mathsf{True})\Big)
\end{aligned}$$

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.
- When we elide stack s in WP-rules, like $\mathsf{wp}\, e\, \{\Phi\}$, then the rules holds for all such WP with any same s. This is useful in making local WP rules look less cluttered.

Laws of weakest precondition. The rules in figure 2.2 and figure 2.2 can all be derived.

WP-VALUE

$$\Phi(v) \vdash \mathsf{wp}_{\mathcal{E}}\, v, s\,\{\Phi\}$$

WP-ATOMIC

$$\frac{\mathsf{atomic}(e)}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e, s \left\{x.\ {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} P\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e, s\,\{x.\, P\}}$$

WP-STRONG-MONO

$$\frac{\mathcal{E}_1 \subseteq \mathcal{E}_2}{(\forall v.\, \Phi(v) \Rrightarrow_{\mathcal{E}_2} \Psi(v)) * \mathsf{wp}_{\mathcal{E}_1}\, e\,\{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}_2}\, e\,\{\Psi\}}$$

FUPD-WP

$$\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\}$$

WP-FUPD

$$\mathsf{wp}_{\mathcal{E}}\, e\,\{x.\, \Rrightarrow_{\mathcal{E}} \Phi(x)\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\}$$

Figure 2: General WP rules

WP-SKIP

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

WP-ASSIGN

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

WP-RET

$$\mathsf{wp}_{\mathcal{E}}\, k'(v), (KS, \Omega_1)\,\{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, k(\mathtt{return}\, v), (K_{call}(k', \Omega_1) :: KS, \Omega_2)\,\{\Phi\}$$

WP-BIND

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

WP-OP

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

WP-LOAD

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

Figure 3: WP rules specific to Iris$^{\mathcal{C}}$, part I

2.3 Soundness

The soundness of WP-style program is proven by showing that it is *adequate*:

Lemma 2.1. *For all $e, s, \sigma, \Phi : Val \to \mathsf{Prop}$,*

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

WP-SEQ

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

WP-ALLOC

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

WP-IF-TRUE

$$\triangleright\, \mathsf{wp}_{\mathcal{E}}\, e_1, s\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{if}(\mathsf{true})\, \{e_1\}\, \mathtt{else}\, \{e_2\}, s\, \{\Phi\}$$

WP-IF-FALSE

$$\triangleright\, \mathsf{wp}_{\mathcal{E}}\, e_2, s\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{if}(\mathsf{false})\, \{e_1\}\, \mathtt{else}\, \{e_2\}, s\, \{\Phi\}$$

WP-FST

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

WP-SND

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

WP-CALL

$$f \mapsto_{\text{TEXT}} \mathit{Function}(\tau, ps, e) * \triangleright(\mathsf{wp}_{\mathcal{E}}\, e[ps/ls], (K_{call}(k, \Omega') :: KS, \Omega)\, \{\Phi\}) \vdash \mathsf{wp}_{\mathcal{E}}\, k(f(ls) : \tau), (KS, \Omega')\, \{\Phi\}$$

WP-CAS-FAIL

$$\frac{\vdash_{\text{typeof}} v_1 : \tau \qquad \vdash_{\text{typeof}} v_2 : \tau \qquad v_1 \neq v}{\triangleright\, l \mapsto_q v : \tau * \triangleright(l \mapsto_q v : \tau \mathrel{-\!\!*} \Phi(\mathsf{false})) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{CAS}(l, v_1, v_2) : \tau, s\, \{\Phi\}}$$

WP-CAS-SUC

$$\frac{\vdash_{\text{typeof}} v_1 : \tau \qquad \vdash_{\text{typeof}} v_2 : \tau}{\triangleright\, l \mapsto_q v_1 : \tau * \triangleright(l \mapsto_q v_2 : \tau \mathrel{-\!\!*} \Phi(\mathsf{false})) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{CAS}(l, v_1, v_2) : \tau, s\, \{\Phi\}}$$

WP-FORK

$$f \mapsto_{\text{TEXT}} \mathit{Function}(\tau, \emptyset, e) * \triangleright\, \Phi(\mathtt{void}) * \triangleright(\mathsf{wp}\, e, (\emptyset, \emptyset)\, \{\_.\, \mathsf{True}\}) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathtt{fork}(f() : \tau) vs, s\, \{\Phi\}$$

WP-WHILE

$$\triangleright\, \mathsf{wp}_{\mathcal{E}}\, \mathtt{if}(c)\, \{e; \mathtt{continue}\}\, \mathtt{else}\, \{\mathtt{break}\}, (K_{while}(c, e, k) :: KS, \Omega)\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, k(\mathtt{while}(c)\, \{e\}, (KS, \Omega)\, \{\Phi\}$$

WP-BREAK

$$\mathsf{wp}_{\mathcal{E}}\, k'(\mathtt{void}), (KS, \Omega)\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, k(\mathtt{break}), (K_{while}(c, e, k') :: KS, \Omega)\, \{\Phi\}$$

WP-CONTINUE

$$\mathsf{wp}_{\mathcal{E}}\, k'(\mathtt{while}(c)\, \{e\}), (KS, \Omega)\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, k(\mathtt{continue}), (K_{while}(c, e, k') :: KS, \Omega)\, \{\Phi\}$$

Figure 4: WP rules specific to Iris$^{\mathcal{C}}$, part II

3 Extensions

Besides the core language and its logic, we built several extension on top of it in a compatible way.

3.1 Array

Using the most primitive form of aggregated structure, product type, we can easily derive array (implemented) or even `struct` in future work.

"A pointer p pointing to an array vs of n elements which are τ-typed" is expressed with:

$$p \mapsto_q vs : \tau^n$$

In implementation, vs is `varray (l:list val)`, and τ^n is `tyarray (t:type) (n:nat)`.

We also have an intermediate representation of a slice of array:

$$p \mapsto_q [v_i : \tau, ..., v_{i+l-1} : \tau]$$

The first form is easier for allocation, since it is defined as a unified aggregate structure; but the second form is easier for use, for example, splitting or indexing. We have the following lemmas proven in Coq:

$$\text{SPLIT-SLICE} \quad p \mapsto_q [v_i : \tau, ..., v_{i+l_1-1} : \tau] * p \mapsto_q [v_{i+l_1} : \tau, ..., v_{i+l_1+l_2-1} : \tau] \equiv p \mapsto_q [v_i : \tau, ..., v_{i+l_1+l_2-1} : \tau]$$

$$\text{INDEX-SPEC} \quad \frac{i < n}{\{p \mapsto_q vs : \tau^n\} \; !_\tau \, p + i \, \{v. \, p \mapsto_q vs : \tau^n * vs \mathbin{!!} i = \texttt{Some}(v)\}}$$

4 Automation

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

4.1 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_alloc`, `wp_op`, `wp_fst`, `wp_snd`, `wp_cas_fail`, `wp_cas_suc`, `wp_let` are similar to the one above
- `wp_skip`: skip over any value before a sequencing operator
- `wp_run`: keep executing as long as it can, which feels like doing symbolic execution automatically
- `wp_ret`: return if the head expression is already value

4.2 Tactics for Algebraic Simplification

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

Refine RA. `rewrite_op_cfgs`: Simplify product of the configuration list component.

4.3 Misc Tactics

Inversion. `inversion_estep`, `inversion_cstep_as`, and `inversion_jstep_as` are all designed to automatically match stepping relation assumption and give proper names to the important results produced by `inversion`.

Evaluation Context Equality. `gen_eq <H> <E1> <E2> <KS>` can generate equalities between expressions `E1` and `E2`, and between `KS` and empty context. It assumes that both `E1` and `E2` are normal form, and there is equality between the filled ones: `fill_ectxs E1 KS = E2`.

5 Refinement

5.1 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 $(\tilde{c}, \tilde{\sigma}) \rightsquigarrow_{spec} (\tilde{c}', \tilde{\sigma}')$ (as well as derived $(\tilde{c}, \tilde{\sigma}) \rightsquigarrow^{*}_{spec} (\tilde{c}', \tilde{\sigma}')$).

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

5.2 Refinement RA

We defined a new RA $\textsc{Refine}$ to capture the history of spec code execution:

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

The validity of $\textsc{Refine}$:

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

The multiplication of $\textsc{View}$ and $\textsc{Refine}$ is defined as:

$$\begin{aligned} \mathsf{master} \cdot \_ &\triangleq \mathsf{master} \\ \_ \cdot \mathsf{master} &\triangleq \mathsf{master} \\ \mathsf{snapshot} \cdot \mathsf{snapshot} &\triangleq \mathsf{snapshot} \end{aligned}$$

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

The disjointness of multiplication is refined as:

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

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

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

5.3 Refinement Ghost State, Invariant and Rules

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

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

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

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

5.4 Soundness

First we define the simulation relationship between the *Expr* and SpecCode:

$$v \Downarrow \mathtt{done}(v) \qquad\qquad \frac{(e, \sigma, \sigma_l) \rightsquigarrow (e', \sigma', \sigma_l') \quad (\tilde{c}, \tilde{\sigma}) \rightsquigarrow^{*}_{spec} (\tilde{c}', \tilde{\sigma}')}{e \Downarrow c}$$

Our final soundness lemma, which is proven in Coq, is like below (the step indexing part is simplified away):

$$\frac{(e \mathbin{+\!\!+} t_1, \sigma) \rightsquigarrow^{n}_{c} ((v, s_l) \mathbin{+\!\!+} t_2, \sigma')}{\text{world}(\sigma) * (\boxed{I_{\textsc{Refine}}}^{\iota} * \text{sstate}(\tilde{\sigma}) * \text{scode}(\tilde{c})) * \mathsf{wp}\, e \,\{v.\, \text{sstate}(\tilde{\sigma}') * \text{scode}(\mathtt{done}(v))\} * \text{wptp}(t_1) \vdash \overset{n+2}{\triangleright}\, v \Downarrow c}$$