Iris-OS Documentation

Zhen Zhang

April 10, 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 at the document and the Coq formalization in the git repository 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 environement 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 ($(e, \sigma) \to_c (e', \sigma')$) pointwise.

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

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:

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

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,

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

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

The definition of WP is exactly the same as in Iris's program logic. In fact, I instantiated this part in Coq instead of reinventing the wheel.

Here are some conventions:

- If we leave away the mask, we assume it to default to $\top$.
- 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-SKIP
$$\triangleright\, \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, v; e\, \{\Phi\}$$

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

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

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

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

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

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

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

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

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

2.3 Soundness

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

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

3 OS and Virtualization

This section discusses the system features as part of the formalization. The first part, OS, mainly include interrupt and simulation. The second part, virtualization, mainly include virtual memory model and the memory allocator on top of that.

3.1 Interrupt

Interrupt is a hardware features, thus we will only write down its abstract specification without proving it. However, in the OS's perspective, multi-level interrupt will enable us to open/close OS-global invariants for different level of priorities.

3.2 Simulation

Simulation piggybacks on the WP-based program logic through two special kinds of assertions: spec state witnessing assertion (sstate($\tilde{\sigma} : [X \hookrightarrow v]$)) and spec code assertion (scode($\tilde{c}$), more defined below).

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

Then we define specification stepping relationship $(\tilde{c}, \tilde{\sigma}) \to_{spec} (\tilde{c}', \tilde{\sigma}')$:

$$\textsc{ss-rel}\ \frac{\tilde{\sigma_f} \perp \tilde{\sigma} \quad r(\tilde{\sigma}, v, \tilde{\sigma}')}{(\texttt{rel}(r), \tilde{\sigma_f} \cup \tilde{\sigma}) \to_{spec} (\texttt{done}(v), \tilde{\sigma_f} \cup \tilde{\sigma}')}$$

We can also define the reflective transitive closure $(\tilde{c}, \tilde{\sigma}) \to^*_{spec} (\tilde{c}', \tilde{\sigma}')$ trivially.

With this, we can give refinement style proofs for certain kernel APIs. The following proof rule is proved in Coq:

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

3.3 VMM

We will first write down the abstract spec for page table primitives: `insert`, `lookup`, and `delete`. They will change the locally owned page table content, which is modeled by a `gmap`.

On top of that, supposing that the physical heap memory is shared across the processes, and our language provides related operations. We can combine these operation with page table access to virtualize an exclusively owned, sequentially allocated memory space. The relevant operations will be `mem_init`, `mem_read` and `mem_write`.

Since our final client is a high-level language, we want to expose an object allocator interface built on top of the local virtual memory. The new operations for runtime to use will be `mem_alloc`, `mem_free`. `mem_init` will be called at the initialization time of the system.

4 Misc

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

4.1 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

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

Theorem 4.1. *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 4.2. *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*.

Here we prove two lemmas about the syntactic structure on paper:

Lemma 4.3. $\forall e : \mathit{Expr}, k : \text{Cont}.\, \text{is\_enf}(e) \to \text{unfill}(k(e)) = (k, e)$

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 4.4. *Induction scheme for Expr:*

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

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. □