Iris-OS

Zhen Zhang

March 30, 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 Introduction

1.1 Background

1.2 Master Plan

Our work explores two dimensions of mechanized OS verification:

- Language-level guarantee through trused virtualization i.e. a verified *Unikernel*
- C-like language verification framework in Coq based on Iris

1. Unikernel. In this part, we will design a new unikernel architecture. The high-level languague will be F⋆, compiled to C with Kremlin, and linked with a minimal substrate providing booting etc.

We don't currently have any real implementation of such system, but it should not be very difficult to build. What we care about it, instead, is the **linking** part. For example, to provide heap allocation, we will expose a malloc-like C interface to F⋆. We can write some properties about these interfaces in F⋆, which should be preserved in the lower level. Also, to show that each guest OS on the hypervisor won't interfere with each other, we want to do some meta-theoretic proofs of the whole system in Coq as well.

In this work, we will focus on the memory management part, and prove related lemmas for such a perspective system.

2. Iris for C In this part, we will build a new Coq framework specific for languages similar to C, and designed for verifying OSes. While the part I is more about modelling, specifying and "real" proofs, this part is more about the tooling for this work and more future users.

What are particularly interesting in this are:

- Compared functional languages, which Iris usually instantiates to, C-like lanugage is harder since it can `return`, `break` and even `throw` (or `longjmp`, if you insist). How can we adapt the WP-based program logic to such exotic features? Can we make the adaption process itself easier?
- Can Iris make it easier to provide more advanced features such as concurrency, function pointer (higher-order) etc.?
- When we don't need that much fine-grained concurrency, can we improve the level of proof automation?
- For OS verification, contextual refinement is vital for specifying APIs. Can we do that too?

Though we can't promise to answer solve all questions, we should make sure that our program logic is sound.

1.3 Related Work

2 Language

2.1 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 $o \in \mathbb{Z}^+$.

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

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.

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

2.2 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.

$$\text{ES-BINOP}\ \frac{[\![oplus]\!](v_1, v_2) = v'}{(v_1 \oplus v_2, \sigma) \to_{Expr} (v', \sigma)} \qquad \text{ES-DEREF-TYPED}\ \frac{\vdash_{\text{tychk}} v : \tau \quad \sigma \vdash l \mapsto \text{encode}(v)}{(!_\tau\, l, \sigma) \to_{Expr} (v, \sigma)} \qquad \text{ES-FST}\ ((v_1, v_2).1, \sigma) \to_{Expr} (v_1, \sigma)$$

$$\text{ES-SND}\ ((v_1, v_2).2, \sigma) \to_{Expr} (v_2, \sigma) \qquad \text{ES-ASSIGN}\ (l \leftarrow v, \sigma) \to_{Expr} (\texttt{skip}, \sigma[l \mapsto \text{encode}(v)]) \qquad \text{ES-SEQ}\ (v; e, \sigma) \to_{Expr} (e, \sigma)$$

$$\text{ES-ALLOC}\ \frac{\vdash_{\text{tychk}} v : \tau \quad \forall o'.\, \sigma(b, o') = \bot}{(\texttt{alloc}_\tau(v), \sigma) \to_{Expr} (b\#o, \sigma[b\#o \mapsto v])} \qquad \text{ES-WHILE-TRUE}\ (\texttt{while}_e(\mathsf{true})\,\{s\}, \sigma) \to_{Expr} (s; \texttt{while}_e(e)\,\{s\}, \sigma)$$

$$\text{ES-WHILE-FALSE}\ (\texttt{while}_e(\mathsf{false})\,\{s\}, \sigma) \to_{Expr} (\texttt{skip}, \sigma) \qquad \text{ES-BIND'}\ \frac{\text{is\_jump}(e) = \mathsf{False} \quad (e, \sigma) \to_{Expr} (e', \sigma')}{((k :: ks)e, \sigma) \to_{Expr} ((k :: ks)e', \sigma')}$$

Figure 1: Semantics rules

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

XXX: Define $\sigma \vdash l \mapsto v \overset{\rightarrow}{{}_{\text{byte}}}$

2.3 Type System and Environment

Local Typing Rules. The types are defined in §2.1, 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.

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

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

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)

3 Program Logic

This section describes how to build a program logic for the C language (*c.f.* §2) on top of the base logic of Iris.

3.1 Extended Assertions

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

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

3.2 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,

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

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

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.

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

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

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

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

4 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.

4.1 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.

4.2 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 refinment style proofs for certain kernel APIs. The following proof rule is proved in Coq:

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

4.3 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 modelled 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.

5 Misc

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

5.1 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 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*.

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

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

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 5.4. *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. □