Mox Notes

Jules Jacobs

November 12, 2025

1 Modes

We distinguish two mode families—*past* and *future*. Each axis is ordered by the conversion relation $\leq_{\text{to}}$ (values that can be coerced "to" a weaker mode) and an in-placement relation $\leq_{\text{in}}$ (values that can live "in" a surrounding mode). For future modes the two orders coincide, whereas for past modes the in-placement relation reverses conversion.

Past axes (p)

uniqueness (u):	UNIQUE $\leq_{\text{to}}$ ALIASED
	ALIASED $\leq_{\text{in}}$ UNIQUE
contention (c):	UNCONTENDED $\leq_{\text{to}}$ SHARED $\leq_{\text{to}}$ CONTENDED
	CONTENDED $\leq_{\text{in}}$ SHARED $\leq_{\text{in}}$ UNCONTENDED

Future axes (f)

linearity (l):	MANY $\leq_{\text{to}}$ ONCE $\leq_{\text{to}}$ NEVER
	MANY $\leq_{\text{in}}$ ONCE $\leq_{\text{in}}$ NEVER
portability (p):	PORTABLE $\leq_{\text{to}}$ NON-PORTABLE
	PORTABLE $\leq_{\text{in}}$ NON-PORTABLE
areality (a):	GLOBAL $\leq_{\text{to}}$ REGIONAL $\leq_{\text{to}}$ LOCAL
	GLOBAL $\leq_{\text{in}}$ REGIONAL $\leq_{\text{in}}$ LOCAL

J: It isn't clear that the linearity axis' $\leq_{\text{in}}$ relation is useful for anything, at least at the moment.

J: Carefully check the $\leq_{\text{in}}$ relation, considering the borrowing issue.

We collect the past modes into the tuple $p = (\text{uniqueness}, \text{contention})$ and the future modes into $f = (\text{areality}, \text{linearity}, \text{portability})$, lifting $\leq_{\text{to}}$ and $\leq_{\text{in}}$ componentwise. A mode is then the pair $m = (p, f)$, with both relations lifted again to m.

Intuition. Modes are about *deep* operations on values. For example, the uniqueness and linearity axes are about the operation of creating an alias to a value.

Alias creation is deep in the sense that when we alias a data structure, then all its children should also be considered aliased (i.e., there are multiple pointer paths from the stack roots to the element). Uniqueness is about whether the pointer path to it is unique, i.e., whether an aliasing operation has been performed on it in the past, and linearity is about whether we're allowed to perform an aliasing operation on it in the future.

A UNIQUEly referenced value can be coerced to an ALIASED value – we simply forget that it was uniquely referenced – hence UNIQUE $\leq_{\text{to}}$ ALIASED. It is also safe to store an aliased value inside a uniquely referenced structure, so ALIASED $\leq_{\text{in}}$ UNIQUE. Conversely, it is nonsensical to say a uniquely referenced value is stored in an aliased container, because aliasing is a deep property: if the container is considered aliased, then all its elements are as well.

For future modes we additionally track the sentinel NEVER on the linearity axis. A NEVER function cannot be invoked; operationally it represents a closure whose call capability has been consumed. Aliasing a ONCE closure demotes it to NEVER rather than rejecting the program outright.

If an aliasing operation may be performed on a value (mode MANY, think "aliasable"), it is safe to coerce it to a value on which we are not allowed to perform an aliasing operation (mode ONCE, think "nonaliasable"), hence MANY $\leq_{\text{to}}$ ONCE. Conversely, values that we can create aliases to may be safely stored in containers that we are not allowed to create aliases to, hence MANY $\leq_{\text{in}}$ ONCE. Allowing the converse storage would be unsound: placing a one-time value inside a aliasable container would expose the value to multiple aliases by aliasing the outer container.

In summary, the guarantees along past axes can get weaker as one goes from a container to its elements, and the restrictions along future axes also get weaker as one goes from a container to its elements. The flipping of the $\leq_{\text{to}}$ order stems from the fact that it's safe to *forget* facts about the past whereas it's safe to *add* restrictions on the future.

2 Expressions

$e ::=$	– expressions
$\mid x \mid \textbf{let}\, x = e_1\, \textbf{in}\, e_2$	– variables
$\mid \textbf{unit} \mid \textbf{absurd}\, e$	– unit and empty
$\mid e_1\, e_2 \mid \textbf{fun}\, x \Rightarrow e$	– functions
$\mid (e_1, e_2) \mid \textbf{let}\, (x_1, x_2) = e_1\, \textbf{in}\, e_2$	– pairs
$\mid \textbf{left}(e) \mid \textbf{right}(e)$	– sums
$\mid \textbf{match}\, e\, \textbf{with}\, \textbf{left}(x_1) \Rightarrow e_1 \mid \textbf{right}(x_2) \Rightarrow e_2$	
$\mid \textbf{ref}\, e \mid\, !e \mid e_1 := e_2$	– references

3 Types

$$
\begin{array}{ll}
\tau ::= & \\
\quad \mid \mathbf{unit} \mid \mathbf{empty} & \text{– unit and empty} \\
\quad \mid \tau_1 \xrightarrow[f]{} \tau_2 & \text{– functions} \\
\quad \mid \tau_1 \underset{s}{\times} \tau_2 & \text{– pairs} \\
\quad \mid \tau_1 \underset{s}{+} \tau_2 & \text{– sums} \\
\quad \mid \mathbf{ref}_r\, \tau & \text{– references}
\end{array}
$$

Notation Function arrows carry a function (future) mode annotation f, drawing from the future lattice so $f = (\text{areality}, \text{linearity}, \text{portability})$. Storage annotations s appear on products and sums, recording how elements are kept with $s = (\text{uniqueness}, \text{areality})$. Reference annotations r decorate mutable references and collect the axes that matter for cells: $r = (\text{uniqueness}, \text{areality}, \text{contention})$ and we write $r_{\mathrm{u}}, r_{\mathrm{a}}, r_{\mathrm{c}}$ for their respective components. We write $\hat{f}$ for the embedding of f into the full mode lattice (past component $\bot_{\mathrm{in}}$, future component f), and $\hat{s}$ and $\hat{r}$ for the embeddings of s and r respectively. Comparisons such as $s_1 \leq_{\mathrm{to}} s_2$ or $r_1 \leq_{\mathrm{in}} r_2$ are taken componentwise, and joins and meets on s or r use the $\leq_{\mathrm{in}}$ order on each axis; we reuse the symbols $\sqcup$ and $\sqcap$ for these operations.

Mutable references The type $\mathbf{ref}_r\, \tau$ denotes a mutable cell that stores a value of type τ. The uniqueness component r_{u} tracks how many aliases exist to the cell's handle; the areality component r_{a} tracks where the cell can reside; and the contention component r_{c} records how much simultaneous access the cell tolerates. Creating a reference defaults to the most precise handle $r_0 = (\textsc{unique}, \textsc{local}, \textsc{uncontended})$, from which weaker capabilities can be derived via subtyping or aliasing.

4 Kinding Rules

We write $\tau : m$ to state that type τ is well-formed at mode m. For storage annotations we write $m \sqcap \hat{s}$ for the meet of m with $\hat{s}$ in the $\leq_{\mathrm{in}}$-lattice; this leaves the axes not mentioned in s untouched and meets the uniqueness and areality axes with those recorded by s. For reference annotations we analogously write $m \sqcap \hat{r}$, meeting the uniqueness, contention, and areality axes with the capabilities recorded by r. Here $\top_{\mathrm{in}}$ denotes the greatest mode with respect to $\leq_{\mathrm{in}}$.

$$\frac{}{\mathbf{unit} : m} \qquad \frac{}{\mathbf{empty} : m} \qquad \frac{\hat{f} \leq_{\text{in}} m \qquad \tau_1 : \top_{\text{in}} \qquad \tau_2 : \top_{\text{in}}}{\tau_1 \underset{\hat{f}}{\to} \tau_2 : m}$$

$$\frac{\hat{s} \leq_{\text{in}} m \qquad \tau_1 : m \sqcap \hat{s} \qquad \tau_2 : m \sqcap \hat{s}}{\tau_1 \underset{s}{\times} \tau_2 : m} \qquad \frac{\hat{s} \leq_{\text{in}} m \qquad \tau_1 : m \sqcap \hat{s} \qquad \tau_2 : m \sqcap \hat{s}}{\tau_1 \underset{s}{+} \tau_2 : m}$$

$$\frac{\hat{r} \leq_{\text{in}} m \qquad \tau : m \sqcap \hat{r}}{\mathbf{ref}_r\, \tau : m}$$

J: Carefully check the kinding rules

J: Instead of kinding rules, consider baking this into the typing rules. This is more flexible, because we can always construct the type, we just cannot use it if it is ill-kinded.

We will omit the annotations when they are not needed.

5 Typing Rules

Typing judgments use linear contexts: $\Gamma_1 \uplus \Gamma_2$ denotes a partition of the context into disjoint subcontexts, and $\emptyset$ is the empty context.

Variables

$$\frac{}{x : \tau \vdash x : \tau} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau_2}$$

Unit and Empty

$$\frac{}{\emptyset \vdash \mathbf{unit} : \mathbf{unit}} \qquad \frac{\Gamma \vdash e : \mathbf{empty}}{\Gamma \vdash \mathbf{absurd}\, e : \tau}$$

Functions

$$\frac{🔒_f(\Gamma), x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}\, x \Rightarrow e : \tau_1 \underset{\hat{f}}{\to} \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2 \vdash e_1\, e_2 : \tau_2}$$

Pairs

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma_2, x_1 : \tau_1, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{let}\, (x_1, x_2) = e_1\, \mathbf{in}\, e_2 : \tau}$$

Sums

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{left}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{right}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma_0 \vdash e : \tau_1 + \tau_2 \qquad \Gamma_1, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma_1, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma_0 \uplus \Gamma_1 \vdash \mathbf{match}\, e\, \mathbf{with}\, \mathbf{left}(x_1) \Rightarrow e_1 \mid \mathbf{right}(x_2) \Rightarrow e_2 : \tau}$$

References

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ref}\, e : \mathbf{ref}\, \tau} \qquad \frac{\Gamma \vdash e : \mathbf{ref}\, \tau}{\Gamma \vdash\, !e : \tau} \qquad \frac{\Gamma_1 \vdash e_1 : \mathbf{ref}\, \tau \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash e_1 := e_2 : \mathbf{unit}}$$

We write simply $\mathbf{ref}\, e$ when allocating a cell; the resulting handle adopts the most precise capability $r_0 = (\textsc{unique}, \textsc{local}, \textsc{uncontended})$, with weaker modes obtained by subtyping its type. A dereference yields the stored payload, and assignment consumes the contexts used to establish the reference and the incoming value.

Subsumption

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \sqsubseteq \tau_2}{\Gamma \vdash e : \tau_2}$$

Aliasing

$$\frac{\Gamma, x : \tau_1', x : \tau_1' \vdash e : \tau_2 \qquad \tau_1 \underset{\text{alias}}{\rightsquigarrow} \tau_1'}{\Gamma, x : \tau_1 \vdash e : \tau_2}$$

6 Subtyping Rules

Base

$$\frac{}{\mathbf{unit} \sqsubseteq \mathbf{unit}} \qquad \frac{}{\mathbf{empty} \sqsubseteq \mathbf{empty}}$$

Functions

$$\frac{\tau_1' \sqsubseteq \tau_1 \qquad \tau_2 \sqsubseteq \tau_2' \qquad f_1 \leq_{\text{to}} f_2}{\tau_1 \xrightarrow[f_1]{} \tau_2 \sqsubseteq \tau_1' \xrightarrow[f_2]{} \tau_2'}$$

Products and Sums

$$\frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2' \qquad s_1 \leq_{\text{to}} s_2}{\tau_1 \underset{s_1}{\times} \tau_2 \sqsubseteq \tau_1' \underset{s_2}{\times} \tau_2'} \qquad \frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2' \qquad s_1 \leq_{\text{to}} s_2}{\tau_1 \underset{s_1}{+} \tau_2 \sqsubseteq \tau_1' \underset{s_2}{+} \tau_2'}$$

References

$$\frac{r_1 \leq_{\text{to}} r_2}{\mathbf{ref}_{r_1}\,\tau \sqsubseteq \mathbf{ref}_{r_2}\,\tau}$$

The payload type is invariant because references support mutation.

7 Aliasing

The judgment $\tau \underset{\text{alias}}{\rightsquigarrow} \tau'$ records that aliasing τ is allowed and yields type τ'.

Base

$$\frac{}{\mathbf{unit} \underset{\text{alias}}{\rightsquigarrow} \mathbf{unit}} \qquad \frac{}{\mathbf{empty} \underset{\text{alias}}{\rightsquigarrow} \mathbf{empty}}$$

Functions

$$\frac{f = (a, \text{MANY}, p)}{\tau_1 \underset{f}{\rightarrow} \tau_2 \underset{\text{alias}}{\rightsquigarrow} \tau_1 \underset{f}{\rightarrow} \tau_2}$$

$$\frac{f = (a, l, p) \qquad l \in \{\text{NEVER}, \text{ONCE}\} \qquad f' = (a, \text{NEVER}, p)}{\tau_1 \underset{f}{\rightarrow} \tau_2 \underset{\text{alias}}{\rightsquigarrow} \tau_1 \underset{f'}{\rightarrow} \tau_2}$$

Products and Sums

$$\frac{\tau_1 \underset{\text{alias}}{\rightsquigarrow} \tau_1' \qquad \tau_2 \underset{\text{alias}}{\rightsquigarrow} \tau_2' \qquad s = (u, a)}{\tau_1 \underset{s}{\times} \tau_2 \underset{\text{alias}}{\rightsquigarrow} \tau_1' \underset{(\text{ALIASED}, a)}{\times} \tau_2'} \qquad \frac{\tau_1 \underset{\text{alias}}{\rightsquigarrow} \tau_1' \qquad \tau_2 \underset{\text{alias}}{\rightsquigarrow} \tau_2' \qquad s = (u, a)}{\tau_1 \underset{s}{+} \tau_2 \underset{\text{alias}}{\rightsquigarrow} \tau_1' \underset{(\text{ALIASED}, a)}{+} \tau_2'}$$

References

$$\frac{\tau \underset{\text{alias}}{\rightsquigarrow} \tau' \qquad r = (u, a, c)}{\mathbf{ref}_r\,\tau \underset{\text{alias}}{\rightsquigarrow} \mathbf{ref}_{(\text{ALIASED}, a, c)}\,\tau'}$$

Aliasing a reference thus forgets both uniqueness and uncontended status while recursively aliasing the stored payload.

8 Locking

Typing a closure of type $\tau_1 \underset{f}{\rightarrow} \tau_2$ applies a lock to the ambient context, written $🔒_f(\Gamma)$. Here $f = (\text{areality}, \text{linearity}, \text{portability})$ tracks the function mode. The lock records which bindings remain accessible inside the closure and how their

modes are weakened so that the body can run safely. We define the operation structurally on contexts.

$$(x : \tau') \in \lock_f(\Gamma) \iff (x : \tau) \in \Gamma \ \wedge \ \lock_f(\tau) = \tau'$$

where

$$\begin{aligned}
\lock_f(\mathbf{unit}) &= \mathbf{unit} \\
\lock_f(\mathbf{empty}) &= \mathbf{empty} \\
\lock_f(\tau_1 \xrightarrow[f']{} \tau_2) &= \tau_1 \xrightarrow[f']{} \tau_2 \quad \text{if} \quad f \leq_{\text{to}} f' \\
\lock_f(\tau_1 \underset{s}{\times} \tau_2) &= \lock_f(\tau_1) \underset{s'}{\times} \lock_f(\tau_2) \quad \text{where } s' = (s_{\text{u}} \sqcup f^{\dagger}_{\text{linearity}},\ s_{\text{a}}) \\
\lock_f(\tau_1 \underset{s}{+} \tau_2) &= \lock_f(\tau_1) \underset{s'}{+} \lock_f(\tau_2) \quad \text{where } s' = (s_{\text{u}} \sqcup f^{\dagger}_{\text{linearity}},\ s_{\text{a}}) \\
\lock_f(\mathbf{ref}_r\, \tau) &= \mathbf{ref}_{r'}\, \lock_f(\tau) \quad \text{where } r' = (r_{\text{u}} \sqcup f^{\dagger}_{\text{linearity}},\ r_{\text{a}},\ r_{\text{c}} \sqcup f^{\dagger}_{\text{portability}}) \\
\lock_f(\tau) &= \bot \quad \text{otherwise}
\end{aligned}$$

The dagger map translates closure capabilities into the weakening applied to captured bindings: The linearity and portability axes translate to past modes as follows:

$$\begin{aligned}
\text{ONCE}^{\dagger} &= \text{UNIQUE} & \qquad \text{MANY}^{\dagger} &= \text{ALIASED} \\
\text{NON-PORTABLE}^{\dagger} &= \text{UNCONTENDED} & \qquad \text{PORTABLE}^{\dagger} &= \text{CONTENDED}
\end{aligned}$$

Intuitively, a MANY closure can only reuse captured data as ALIASED, and a PORTABLE closure forces captured bindings to be CONTENDED. When the captured value is a reference we update the pointer's uniqueness and contention components while locking the stored payload recursively. If we lock a type with respect to MANY then we change all UNIQUE to ALIASED. If we lock a type with respect to PORTABLE then we change all UNCONTENDED and SHARED to CONTENDED.

9 Type Inference

We must be able to maintain these constraints on **type variables**:

- Subtyping constraints $\tau_1 \sqsubseteq \tau_2$ between two type variables.
- Alias constraints between two type variables ("type variable τ_1 is the aliased version of type variable τ_2").
- Lock constraints between two type variables ("type variable τ_1 is type variable τ_2 locked with modes f").
- In-placement constraints on a type variable (given by a set of modes m).

We must able able to maintain these constraints on **mode variables**:

- $\leq_{\text{in}}$ constraints between two mode variables.
- $\leq_{\text{to}}$ constraints between two mode variables.
- Domain constraints on mode variables.
- Dagger-constraints between mode variables.

10 Desiderata

1. Borrowing
2. Mode polymorphism
3. Mutable references & Fork
4. Algebraic data types
5. Type inference
6. Modules & abstract types
7. Fine grained uniquness analysis