RustBelt: Securing the Foundations of the Rust Programming Language – Technical appendix

Ralf Jung Jacques-Henri Jourdan Robbert Krebbers Derek Dreyer

November 9, 2017

Contents

1 Syntax

1.1 Grammar

λ_{Rust} is a lambda calculus with natural numbers and state, with explicit deallocation, and with a primitive operation to copy regions of memory. Products and sums are not values, the only exist in their heap representation and are manipulated there, or on a per-field basis.

The grammar of the language is given in Figure 1. e is the grammatical class of expressions and v represents values. Notably, ☣ represents a *poison value*; using poison in any interesting way causes the program to be stuck – basically, anything except loading from and storing to memory. z is any integer, while n and i are natural numbers. Typically, i is used as an index into something (*e.g.*, a list). ℓ is a heap location, their structure will be defined later. x, f are all program variables, the second usually denoting a function. $\overline{x}$ is a list of x (and similar for other metavariables), which can be constructed as $[x_1, x_2, \dots]$. To simplify the formalization of the semantics, we also introduce a notion of *evaluation contexts* K. Memory accesses are annotated with a *memory order* o, which is either non-atomic (**`na`**) or sequentially consistent (**`sc`**). The purpose of this is that the program gets stuck when there are races involving non-atomic accesses. Thus, by proving safety of a program, we show data-race freedom. (The order **`na'`** is just an internal implementation detail.)

This semantics is in some sense too definite, compared to Rust: It fixes the memory representation of products and sums, and it allows mutation of any location at any time.

1.2 Operational semantics

A location $\ell = (i, n)$ consists of a *block* i and an offset into the block n. Allocation and deallocation is always performed on entire blocks. Address arithmetic works within a block: $\ell + m$ increments the offset, and leaves the block number untouched.

A memory h is a finite partial map from locations to pairs of values and lock states:

$$\mathit{Mem} := \mathbb{N} \times \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \mathit{LockSt} \times \mathit{Val}$$

The lock states encode a per-location reader-writer-lock that serves to detect data races. Notice that **`reading`** 0 corresponds to the lock being unlocked. Non-atomic accesses require the location to be locked; they will always be immediately preceded by the operation to acquire that location's lock. Atomic accesses, on the other hand, fail if there is a conflicting lock, *i.e.*, all accesses fail if the write lock is held, and write accesses fail if the read lock is held. Putting all these pieces together, we have shown that a program that is *safe* (*i.e.*, cannot get stuck in any execution) is free of data-races: We can never reach a state such that two different threads will, if they get a chance to take a step now, perform *conflicting* memory accesses—*i.e.*, accesses to the same location, at least one of which is non-atomic, and at least one of which is a write.

To define the behavior of equality tests and CAS, we employ a helper judgment $h \vdash v_1 = v_2$ saying whether v_1 and v_2 can compare (in)equal under memory h. In particular, comparing two different locations of which at least one is not allocated is *non-deterministic*: they can compare equal or unequal. For CAS, this means that the CAS could either succeed or fail. The purpose of O-cas-stuck is to make sure that if such a non-deterministic CAS races with a non-atomic access to the same location, the program is stuck. To this end, if such a situation is detected, we step to the stuck state $0()$.

$$
\begin{aligned}
z &\in \mathbb{Z} \\
\mathit{Expr} \ni e &::= v \mid x \\
&\quad \mid e.e \mid e + e \mid e - e \mid e \leq e \mid e == e \\
&\quad \mid e(\overline{e}) \\
&\quad \mid {}^{*o} e \\
&\quad \mid e_1 :=_o e_2 \\
&\quad \mid \texttt{CAS}(e_0, e_1, e_2) \\
&\quad \mid \texttt{alloc}(e) \\
&\quad \mid \texttt{free}(e_1, e_2) \\
&\quad \mid \texttt{case}\, e\, \texttt{of}\, \overline{e} \\
&\quad \mid \texttt{fork}\, \{\, e \,\} \\
\mathit{Val} \ni v &::= \text{☣} \mid \ell \mid z \mid \texttt{rec}\, f(\overline{x}) := e \\
\mathit{Loc} \ni \ell &::= (i, n) \\
\mathit{Order} \ni o &::= \texttt{sc} \mid \texttt{na} \mid \texttt{na'} \\
\mathit{LockSt} \ni \pi &::= \texttt{writing} \mid \texttt{reading}\, n
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Ctx} \ni K &::= \bullet \\
&\quad \mid K.e \mid v.K \mid K + e \mid v + K \mid K - e \mid v - K \\
&\quad \mid K \leq e \mid v \leq K \mid K == e \mid v == K \\
&\quad \mid K(\overline{e}) \mid v(\overline{v} + [K] + \overline{e}) \\
&\quad \mid {}^{*o} K \mid K :=_o e \mid v :=_o K \\
&\quad \mid \texttt{CAS}(K, e_1, e_2) \\
&\quad \mid \texttt{CAS}(v_0, K, e_2) \\
&\quad \mid \texttt{CAS}(v_0, v_1, K) \\
&\quad \mid \texttt{alloc}(K) \\
&\quad \mid \texttt{free}(K, e_2) \\
&\quad \mid \texttt{free}(e_1, K) \\
&\quad \mid \texttt{case}\, K\, \texttt{of}\, \overline{e}
\end{aligned}
$$

Figure 1: Language syntax.

We use the notation $[<n]$ to denote the set $\{m \mid m < n\}$, and $[\geq m, <n]$ to denote $\{m' \mid m \leq m' < n\}$. The notation $h\,[\ell \leftarrow v]$ denotes the map h updated with location ℓ to map to v. $h\,[\ell \leftarrow v \mid x \in T]$ does the update for every $x \in T$, where ℓ and v may depend on x.

Note how O-alloc initializes memory with ☣ (poison), modeling uninitialized memory. Similarly, instructions that don't actually return any information (like storing) return ☣, ensuring that their return value is not used by the program.

We give the semantics as a small-step reduction relation of machine states, which encompass the current memory and term, written $h \mid e$. Notably, this semantics defines more behaviors than Rust does: The representation of product and sum types is fixed, uninitialized allocated locations have deterministic reads, and it is possible to implement interior mutability without *`UnsafeCell`*.

1.3 Continuation-passing-style let-normal programs

In this section, we define the surface language that will be used for type-checking. To support control flow operators such as `return` or `break`, the type system will enforce program to be in continuation-passing style and in let-normal form. Before we come to the details of this, we need to define some derived constructions that will be primitive in the surface language. In the following, we use f for program variables that are used as functions, and k for program variables used as continuations.

There are some operations that are not primitive to the language in a syntactic sense, but that come with primitive typing rules: Copying a range of memory, and initializing a sum. These operations are pervasively used, but cannot be typed in the type system. So we define them here as derived forms, together with some useful syntactic sugar for sequencing, non-atomic accesses and

$$\boxed{h \vdash v_1 = v_2}$$

$$h \vdash z = z \qquad h \vdash \ell = \ell \qquad \frac{\ell_1 \notin \mathrm{dom}(h) \lor \ell_2 \notin \mathrm{dom}(h)}{h \vdash \ell_1 = \ell_2}$$

$$\boxed{h \vdash v_1 \neq v_2}$$

$$\frac{z_1 \neq z_2}{h \vdash z_1 \neq z_2} \qquad \frac{\ell_1 \neq \ell_2}{h \vdash \ell_1 \neq \ell_2}$$

$$\boxed{h \mid e \rightarrow h' \mid e_1', e_2'^{?}}$$

O-ECTX
$$\frac{h \mid e \rightarrow h \mid e_1', e_2'^{?}}{h \mid K[e] \rightarrow h \mid K[e_1'], e_2'^{?}}$$

O-PROJ
$$h \mid \ell.n \rightarrow h \mid \ell + n$$

O-ADD
$$\frac{z_1 + z_2 = z'}{h \mid z_1 + z_2 \rightarrow h \mid z'}$$

O-SUB
$$\frac{z_1 - z_2 = z'}{h \mid z_1 - z_2 \rightarrow h \mid z'}$$

O-LE-TRUE
$$\frac{z_1 \leq z_2}{h \mid z_1 \leq z_2 \rightarrow h \mid 1}$$

O-LE-FALSE
$$\frac{z_1 > z_2}{h \mid z_1 \leq z_2 \rightarrow h \mid 0}$$

O-EQ-TRUE
$$\frac{h \vdash v_1 = v_2}{h \mid v_1 == v_2 \rightarrow h \mid 1}$$

O-EQ-FALSE
$$\frac{h \vdash v_1 \neq v_2}{h \mid v_1 == v_2 \rightarrow h \mid 0}$$

O-ALLOC
$$\frac{n > 0 \qquad \ell = (i, n') \qquad \{i\} \times \mathbb{N} \mathrel{\#} \mathrm{dom}(h) \qquad h' = h\,[\ell + m \leftarrow (\texttt{reading}\,0, \text{☣}) \mid m \in [<n]]}{h \mid \texttt{alloc}(n) \rightarrow h' \mid \ell}$$

O-FREE
$$\frac{n > 0 \qquad \ell = (i, n') \qquad \mathrm{dom}(h) \cap \{i\} \times \mathbb{N} = \{i\} \times ([\geq n', < n' + n]) \qquad h' = h\,[\ell + m \leftarrow \bot \mid m \in [<n]]}{(h \mid \texttt{free}(n, \ell)) \rightarrow (h' \mid \text{☣})}$$

O-DEREF-SC
$$\frac{h(\ell) = (\texttt{reading}\,n, v)}{h \mid {}^{*\texttt{sc}}\ell \rightarrow h \mid v}$$

O-DEREF-NA
$$\frac{h(\ell) = (\texttt{reading}\,n, v)}{(h \mid {}^{*\texttt{na}}\ell) \rightarrow (h\,[\ell \leftarrow (\texttt{reading}\,n+1, v)] \mid {}^{*\texttt{na'}}\ell)}$$

O-DEREF-NA'
$$\frac{h(\ell) = (\texttt{reading}\,n+1, v)}{(h \mid {}^{*\texttt{na'}}\ell) \rightarrow (h\,[\ell \leftarrow (\texttt{reading}\,n, v)] \mid v)}$$

O-ASSIGN-SC
$$\frac{h(\ell) = (\texttt{reading}\,0, v')}{(h \mid \ell :=_{\texttt{sc}} v) \rightarrow (h\,[\ell \leftarrow (\texttt{reading}\,0, v)] \mid \text{☣})}$$

O-ASSIGN-NA
$$\frac{h(\ell) = (\texttt{reading}\,0, v')}{(h \mid \ell :=_{\texttt{na}} v) \rightarrow (h\,[\ell \leftarrow (\texttt{writing}, v')] \mid \ell :=_{\texttt{na'}} v)}$$

O-ASSIGN-NA'
$$\frac{h(\ell) = (\texttt{writing}, v')}{(h \mid \ell :=_{\texttt{na'}} v) \rightarrow (h\,[\ell \leftarrow (\texttt{reading}\,0, v)] \mid \text{☣})}$$

O-CAS-FAIL
$$\frac{h(\ell) = (\texttt{reading}\,n, v') \qquad h \vdash v' \neq v_1}{(h \mid \texttt{CAS}(\ell, v_1, v_2)) \rightarrow (h \mid 0)}$$

O-CAS-SUC
$$\frac{h(\ell) = (\texttt{reading}\,0, v') \qquad h \vdash v' = v_1}{(h \mid \texttt{CAS}(\ell, v_1, v_2)) \rightarrow (h\,[\ell \leftarrow (\texttt{reading}\,0, z_2)] \mid 1)}$$

O-CAS-STUCK
$$\frac{h(\ell) = (\texttt{reading}\,n, v') \qquad n > 0 \qquad h \vdash v' = v_1}{(h \mid \texttt{CAS}(\ell, v_1, v_2)) \rightarrow (h \mid 0())}$$

O-CASE
$$(h \mid \texttt{case}\,i\,\texttt{of}\,\overline{e}) \rightarrow (h \mid \overline{e}_i)$$

O-APP
$$(h \mid (\texttt{rec}\,f(\overline{x}) := e)(\overline{v})) \rightarrow (h \mid e[\texttt{rec}\,f(\overline{x}) := e/f, \overline{v}/\overline{x}])$$

O-FORK
$$h \mid \texttt{fork}\,\{\,e\,\} \rightarrow h \mid \text{☣}, e$$

Figure 2: Operational semantics.

conditionals.

Finally, we define the operational behavior of the coercions that start and end lifetimes. They don't actually do anything, but they take a physical step – which we need to make the proofs go through.

$$\begin{aligned}
\mathtt{funrec}\, f(\overline{x})\,\mathtt{ret}\, k := e &:= \mathtt{rec}\, f([k] +\!\!+\, \overline{x}) := e \\
\mathtt{let}\, x = e \,\mathtt{in}\, e' &:= (\mathtt{rec}\, \_([x]) := e')(e) \\
e'; e &:= \mathtt{let}\, \_ = e' \,\mathtt{in}\, e \\
\mathtt{letcont}\, k(\overline{x}) := e \,\mathtt{in}\, e' &:= \mathtt{let}\, k = (\mathtt{rec}\, k(\overline{x}) := e) \,\mathtt{in}\, e' \\
\mathtt{jump}\, k(\overline{e}) &:= k(\overline{e}) \\
\mathtt{call}\, f(\overline{e})\,\mathtt{ret}\, k &:= f([k] +\!\!+\, \overline{e})
\end{aligned}$$

$$\begin{aligned}
\mathtt{false} &:= 0 \\
\mathtt{true} &:= 1 \\
\mathtt{if}\, e_0 \,\mathtt{then}\, e_1 \,\mathtt{else}\, e_2 &:= \mathtt{case}\, e_0 \,\mathtt{of}\, [e_1, e_2]
\end{aligned}$$

$$\begin{aligned}
{}^{*}e &:= {}^{*\mathtt{na}}e \\
e_1 := e_2 &:= e_1 :=_{\mathtt{na}} e_2 \\
\mathtt{new} &:= \mathtt{rec}\, \mathit{new}(\mathit{size}) := \\
&\qquad \mathtt{if}\, \mathit{size} == 0 \,\mathtt{then}\, (42, 1337) \,\mathtt{else}\, \mathtt{alloc}(\mathit{size}) \\
\mathtt{delete} &:= \mathtt{rec}\, \mathit{delete}(\mathit{size}, \mathit{ptr}) := \\
&\qquad \mathtt{if}\, \mathit{size} == 0 \,\mathtt{then}\, \text{☣} \,\mathtt{else}\, \mathtt{free}(\mathit{size}, \mathit{ptr}) \\
\mathtt{memcpy} &:= \mathtt{rec}\, \mathit{memcpy}(\mathit{dst}, \mathit{len}, \mathit{src}) := \\
&\qquad \mathtt{if}\, \mathit{len} \leq 0 \,\mathtt{then}\, \text{☣} \,\mathtt{else} \\
&\qquad \mathit{dst}.0 := \mathit{src}.0; \\
&\qquad \mathit{memcpy}(\mathit{dst}.1, \mathit{len} - 1, \mathit{src}.1) \\
e_1 :=_n {}^{*}e_2 &:= \mathsf{memcpy}(e_1, n, e_2) \\
e \overset{\mathsf{inj}\, i}{:==} () &:= e.0 := i \\
e_1 \overset{\mathsf{inj}\, i}{:==} e_2 &:= e_1.0 := i; e_1.1 := e_2 \\
e_1 \overset{\mathsf{inj}\, i}{:==}_n {}^{*}e_2 &:= e_1.0 := i; e_1.1 :=_n {}^{*}e_2
\end{aligned}$$

$$\begin{aligned}
\mathtt{skip} &:= \mathtt{let}\, x = \text{☣} \,\mathtt{in}\, \text{☣} \\
\mathtt{newlft} &:= \text{☣} \\
\mathtt{endlft} &:= \mathtt{skip}
\end{aligned}$$

We distinguish between three classes of expressions: *function bodies F* consist of *instructions I* that operate on *paths p*. The `letcall` operator makes it possible to call *other functions*, passing the remainder of the current function as a continuation. This is in contrast to calling a continuation by just jumping there. We the purpose of $\lambda_{\textsf{Rust}}$, we are not concerned with whole programs, but only with the individual functions of a program.

$$
\begin{aligned}
\mathit{Path} \ni p &::= x \mid p.n \\
\mathit{Instr} \ni I &::= \texttt{false} \mid \texttt{true} \mid z \mid \texttt{funrec}\, f(\overline{x})\,\texttt{ret}\, k := F \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \leq p_2 \\
&\quad \mid \texttt{new}(n) \mid \texttt{delete}(n, p) \mid {}^*p \mid p_1 := p_2 \mid p \overset{\textsf{inj}\, i}{:==} () \mid p_1 \overset{\textsf{inj}\, i}{:==} p_2 \mid p_1 :=_n {}^*p_2 \mid p_1 \overset{\textsf{inj}\, i}{:==}_n {}^*p_2 \\
\mathit{FuncBody} \ni F &::= \texttt{let}\, x = I\, \texttt{in}\, F \mid \texttt{letcont}\, k(\overline{x}) := F_1\, \texttt{in}\, F_2 \mid \texttt{newlft}; F \mid \texttt{endlft}; F \\
&\quad \mid \texttt{if}\, p\, \texttt{then}\, F_1\, \texttt{else}\, F_2 \mid \texttt{case}\, {}^*p\, \texttt{of}\, \overline{F} \mid \texttt{jump}\, k(\overline{x}) \mid \texttt{call}\, f(\overline{p})\, \texttt{ret}\, k
\end{aligned}
$$

1.4 Type System

The key concept of the $\lambda_{\textsf{Rust}}$ type system is the notion of a *lifetime*. Essentially, a lifetime represents a part of the program execution.

Programs are type-checked under five contexts: the *variable context* Γ contains all binders. It assigns variables to their *sort* σ (either program variable $x : \textbf{\textsf{val}}$, a lifetime $\alpha : \textbf{\textsf{lft}}$ or a type $T : \textbf{\textsf{type}}$). Furthermore, there is a context for *external lifetimes* $\mathbf{E}$, a context for *local lifetimes* $\mathbf{L}$, a context assigning *types* to variables $\mathbf{T}$ and a context managing *continuations* $\mathbf{K}$.

$$
\begin{aligned}
\mathit{Sort} \ni \sigma &::= \textbf{\textsf{val}} \mid \textbf{\textsf{lft}} \mid \textbf{\textsf{type}} \\
\Gamma &::= \emptyset \mid \Gamma, X : \sigma \\
\mathit{Lft} \ni \kappa &::= \alpha \mid \textbf{\textsf{static}} \\
\mathbf{E} &::= \emptyset \mid \mathbf{E}, \kappa \sqsubseteq_{\mathrm{e}} \kappa' \\
\mathbf{L} &::= \emptyset \mid \mathbf{L}, \kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} \\
\mathit{Mod} \ni \mu &::= \textbf{\textsf{mut}} \mid \textbf{\textsf{shr}} \\
\mathit{Type} \ni \tau &::= T \mid \textbf{\textsf{bool}} \mid \textbf{\textsf{int}} \mid \lightning_n \\
&\quad \mid \textbf{\textsf{own}}_n\, \tau \mid \&^{\kappa}_{\mu}\, \tau \mid \Sigma\overline{\tau} \mid \Pi\overline{\tau} \mid \forall\overline{\alpha}.\, \textbf{\textsf{fn}}(\digamma : \mathbf{E}; \overline{\tau}) \to \tau \mid \mu\, T.\, \tau \\
\mathbf{T} &::= \emptyset \mid \mathbf{T}, p \lhd \tau \mid \mathbf{T}, p \lhd^{\dagger\kappa} \tau \\
\mathbf{K} &::= \emptyset \mid \mathbf{K}, k \lhd \textbf{\textsf{cont}}(\mathbf{L}; \overline{x}.\, \mathbf{T})
\end{aligned}
$$

Types describe not just single values, but regions of memory – for now, you can think of them as making statements about *lists* of values. The type system introduces an distinction between functions and continuations, matching the program grammar given in §1.3. In particular, continuations are not types. We use this distinction to control that continuations are not passed to other functions or otherwise leaked from their context. We will use function types for Rust-level functions, and continuation types for the basic blocks of a Rust function.

The types of the form $\textbf{\textsf{own}}_n\, \tau$ represent *owned pointers*. The index n gives the size of the block that this object has been allocated in; this is important because only entire blocks can be deallocated.

A particularly interesting type is a *reference* $\&^{\kappa}_{\mu}\, \tau$, also called *(temporarily) borrowed pointer*. References are qualified by a *modifier*, which is either $\textbf{\textsf{mut}}$ (mutable, *i.e.*, unique) and $\textbf{\textsf{shr}}$ (shared). This pointer is only valid as long as its *lifetime* is still active, which can be proven by showing that the lifetime is in the lifetime context $\mathbf{L}$. Functions can be polymorphic over lifetimes.

Finally, the type system supports *recursive types*, with the restriction (enforced by well-formedness) that the recursive occurrence is below a *pointer type*.

The type context can contain “normal” type assignments ($p \lhd \tau$) and type assignments that are *blocked by a lifetime*: $p \lhd^{\dagger\kappa} \tau$ means that we can only use this type assignment again when κ has ended.

1.4.1 Well-formedness

The well-formedness judgments ($\vdash_{\mathsf{wf}}$) document the binding structure of our grammar. There should be no surprises. In the other judgments defined later, we always implicitly assume well-formedness and we also will frequently leave the variable context Γ implicit.

Well-formed paths $\boxed{\Gamma \vdash_{\mathsf{wf}} p}$

$$\frac{x : \textsf{\textbf{val}} \in \Gamma}{\Gamma \vdash_{\mathsf{wf}} x} \qquad \frac{\Gamma \vdash_{\mathsf{wf}} p}{\Gamma \vdash_{\mathsf{wf}} p.n}$$

Well-formed lifetimes $\boxed{\Gamma \vdash_{\mathsf{wf}} \kappa}$

$$\frac{\alpha : \textsf{\textbf{lft}} \in \Gamma}{\Gamma \vdash_{\mathsf{wf}} \alpha} \qquad \Gamma \vdash_{\mathsf{wf}} \textsf{\textbf{static}}$$

Well-formed external lifetime contexts $\boxed{\Gamma \vdash_{\mathsf{wf}} \mathbf{E}}$

$$\Gamma \vdash_{\mathsf{wf}} \emptyset \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \mathbf{L} \qquad \Gamma \vdash_{\mathsf{wf}} \kappa \qquad \Gamma \vdash_{\mathsf{wf}} \kappa'}{\Gamma \vdash_{\mathsf{wf}} \mathbf{L}, \kappa \sqsubseteq_{\mathrm{e}} \kappa'}$$

Well-formed local lifetime contexts $\boxed{\Gamma \vdash_{\mathsf{wf}} \mathbf{L}}$

$$\Gamma \vdash_{\mathsf{wf}} \emptyset \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \mathbf{L} \qquad \Gamma \vdash_{\mathsf{wf}} \kappa \qquad \forall \kappa' \in \overline{\kappa}.\, \Gamma \vdash_{\mathsf{wf}} \kappa'}{\Gamma \vdash_{\mathsf{wf}} \mathbf{L}, \kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa}}$$

Well-formed types $\boxed{\Gamma \vdash_{\mathsf{wf}} \tau}$

$$\frac{T : \textsf{\textbf{type}} \in \Gamma}{\Gamma \vdash_{\mathsf{wf}} T} \qquad \Gamma \vdash_{\mathsf{wf}} \textsf{\textbf{bool}} \qquad \Gamma \vdash_{\mathsf{wf}} \textsf{\textbf{int}} \qquad \Gamma \vdash_{\mathsf{wf}} \lightning\, n \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \tau}{\Gamma \vdash_{\mathsf{wf}} \textsf{\textbf{own}}_n\, \tau} \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \kappa \qquad \Gamma \vdash_{\mathsf{wf}} \tau}{\Gamma \vdash_{\mathsf{wf}} \&_{\mu}^{\kappa}\, \tau}$$

$$\frac{\forall i.\, \Gamma \vdash_{\mathsf{wf}} \overline{\tau}_i}{\Gamma \vdash_{\mathsf{wf}} \Pi\overline{\tau}} \qquad \frac{\forall i.\, \Gamma \vdash_{\mathsf{wf}} \overline{\tau}_i}{\Gamma \vdash_{\mathsf{wf}} \Sigma\overline{\tau}} \qquad \frac{\Gamma, \overline{\alpha}, \digamma : \textsf{\textbf{lft}} \vdash_{\mathsf{wf}} \mathbf{E} \qquad \forall i.\, \Gamma, \overline{\alpha} : \textsf{\textbf{lft}} \vdash_{\mathsf{wf}} \overline{\tau}_i \qquad \Gamma, \overline{\alpha} : \textsf{\textbf{lft}} \vdash_{\mathsf{wf}} \tau}{\Gamma \vdash_{\mathsf{wf}} \forall \overline{\alpha}.\, \textsf{\textbf{fn}}(\digamma : \mathbf{E}; \overline{\tau}) \to \tau}$$

$$\frac{\Gamma, T : \textsf{\textbf{type}} \vdash_{\mathsf{wf}} \tau \qquad T \text{ is guarded by pointer types in } \tau}{\Gamma \vdash_{\mathsf{wf}} \mu T.\, \tau}$$

Well-formed type contexts $\boxed{\Gamma \vdash_{\mathsf{wf}} \mathbf{T}}$

$$\Gamma \vdash_{\mathsf{wf}} \emptyset \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \mathbf{T} \qquad \Gamma \vdash_{\mathsf{wf}} p \qquad \Gamma \vdash_{\mathsf{wf}} \tau}{\Gamma \vdash_{\mathsf{wf}} \mathbf{T}, p \lhd \tau} \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \mathbf{T} \qquad \Gamma \vdash_{\mathsf{wf}} p \qquad \Gamma \vdash_{\mathsf{wf}} \kappa \qquad \Gamma \vdash_{\mathsf{wf}} \tau}{\Gamma \vdash_{\mathsf{wf}} \mathbf{T}, p \lhd^{\dagger\kappa} \tau}$$

Well-formed continuation contexts $\boxed{\Gamma \vdash_{\mathsf{wf}} \mathbf{K}}$

$$\Gamma \vdash_{\mathsf{wf}} \emptyset \qquad \frac{\Gamma \vdash_{\mathsf{wf}} \mathbf{T} \qquad \Gamma \vdash_{\mathsf{wf}} k \qquad \Gamma \vdash_{\mathsf{wf}} \mathbf{L} \qquad \Gamma, \overline{x} : \textsf{val} \vdash_{\mathsf{wf}} \mathbf{T}}{\Gamma \vdash_{\mathsf{wf}} \mathbf{T}, k \lhd \textsf{cont}(\mathbf{L}; \overline{x}.\, \mathbf{T})}$$

1.4.2 Size, Copy, Send, Sync

The *size* of a type says how many memory locations a type spans. It is defined as follows:

$$\begin{aligned}
\mathsf{size}(\textbf{bool}) &:= 1 & \mathsf{size}(\textbf{own}_n\, \tau) &:= 1\\
\mathsf{size}(\textbf{int}) &:= 1 & \mathsf{size}(\&^{\kappa}_{\mu}\, \tau) &:= 1\\
\mathsf{size}(\lightning_n) &:= n & \mathsf{size}(\Pi\overline{\tau}) &:= \sum_i \mathsf{size}(\overline{\tau}_i)\\
& & \mathsf{size}(\Sigma\overline{\tau}) &:= 1 + \max_i \mathsf{size}(\overline{\tau}_i)\\
\mathsf{size}(\mu T.\, \tau) &:= \mathsf{size}(\tau) & \mathsf{size}(\forall\overline{\alpha}.\, \textbf{fn}(\mathsf{F} : \mathbf{E}; \overline{\tau}) \to \tau) &:= 1
\end{aligned}$$

Notice that there is no case for type variables: since well-formed recursive types always have their recursive occurrence below a pointer type, the size of a recursive type does not depend on the size of the recursive occurrence.

Some types are *copyable*, which means they can be used arbitrarily often. This is expressed by the following judgment. Notice that in proving a recursive type to be copyable, you can assume the type variable T to be copy.

Copy types $\boxed{\tau\ \mathsf{copy}}$

$$\textbf{bool}\ \mathsf{copy} \qquad \textbf{int}\ \mathsf{copy} \qquad \lightning_n\ \mathsf{copy} \qquad \&^{\kappa}_{\mathsf{shr}}\, \tau\ \mathsf{copy} \qquad \frac{\forall i.\, \tau_i\ \mathsf{copy}}{\Pi\overline{\tau}\ \mathsf{copy}} \qquad \frac{\forall i.\, \tau_i\ \mathsf{copy}}{\Sigma\overline{\tau}\ \mathsf{copy}}$$

$$(\forall\overline{\alpha}.\, \textbf{fn}(\mathsf{F} : \mathbf{E}; \overline{\tau}) \to \tau)\ \mathsf{copy} \qquad T\ \mathsf{copy} \qquad \frac{\tau\ \mathsf{copy}}{\mu T.\, \tau\ \mathsf{copy}}$$

A type is *send* if ownership can be transferred to another thread. It is *sync* if shared instances of the type can be transferred to another thread.

Send types $\boxed{\tau\ \mathsf{send}}$

$$\textbf{bool}\ \mathsf{send} \qquad \textbf{int}\ \mathsf{send} \qquad \lightning_n\ \mathsf{send} \qquad \frac{\tau\ \mathsf{send}}{\textbf{own}_n\, \tau\ \mathsf{send}} \qquad \frac{\tau\ \mathsf{send}}{\&^{\kappa}_{\mathsf{mut}}\, \tau\ \mathsf{send}} \qquad \frac{\tau\ \mathsf{sync}}{\&^{\kappa}_{\mathsf{shr}}\, \tau\ \mathsf{send}} \qquad \frac{\forall i.\, \tau_i\ \mathsf{send}}{\Pi\overline{\tau}\ \mathsf{send}}$$

$$\frac{\forall i.\, \tau_i\ \mathsf{send}}{\Sigma\overline{\tau}\ \mathsf{send}} \qquad (\forall\overline{\alpha}.\, \textbf{fn}(\mathsf{F} : \mathbf{E}; \overline{\tau}) \to \tau)\ \mathsf{send} \qquad T\ \mathsf{send} \qquad \frac{\tau\ \mathsf{send}}{\mu T.\, \tau\ \mathsf{send}}$$

Sync types $\boxed{\tau\ \textsf{sync}}$

$$\textbf{bool}\ \textsf{sync} \qquad \textbf{int}\ \textsf{sync} \qquad \lightning_n\ \textsf{sync} \qquad \frac{\tau\ \textsf{sync}}{\textbf{own}_n\,\tau\ \textsf{sync}} \qquad \frac{\tau\ \textsf{sync}}{\&^{\kappa}_{\textbf{mut}}\,\tau\ \textsf{sync}} \qquad \frac{\tau\ \textsf{sync}}{\&^{\kappa}_{\textbf{shr}}\,\tau\ \textsf{sync}} \qquad \frac{\forall i.\, \tau_i\ \textsf{sync}}{\Pi\overline{\tau}\ \textsf{sync}}$$

$$\frac{\forall i.\, \tau_i\ \textsf{sync}}{\Sigma\overline{\tau}\ \textsf{sync}} \qquad (\forall\overline{\alpha}.\,\textbf{fn}(\mathsf{F} : \mathbf{E}; \overline{\tau}) \to \tau)\ \textsf{sync} \qquad T\ \textsf{sync} \qquad \frac{\tau\ \textsf{sync}}{\mu T.\, \tau\ \textsf{sync}}$$

1.4.3 Lifetime context judgments

The following judgments express various properties of lifetime contexts.

Lifetime inclusion $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa_1 \sqsubseteq \kappa_2}$

$$\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \textbf{static} \qquad \frac{\kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} \in \mathbf{L} \qquad \kappa' \in \overline{\kappa}}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'} \qquad \frac{\kappa \sqsubseteq_{\mathrm{e}} \kappa' \in \mathbf{E}}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'} \qquad \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa$$

$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa' \qquad \mathbf{E}; \mathbf{L} \vdash \kappa' \sqsubseteq \kappa''}{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa''}$$

Lifetime liveness $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \kappa\ \textsf{alive}}$

$$\mathbf{E}; \mathbf{L} \vdash \textbf{static}\ \textsf{alive} \qquad \frac{\kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} \in \mathbf{L} \qquad \forall i.\, \mathbf{E}; \mathbf{L} \vdash \overline{\kappa}_i\ \textsf{alive}}{\mathbf{E}; \mathbf{L} \vdash \kappa\ \textsf{alive}} \qquad \frac{\mathbf{E}; \mathbf{L} \vdash \kappa\ \textsf{alive} \qquad \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \vdash \kappa'\ \textsf{alive}}$$

Local lifetime context inclusion $\boxed{\Gamma \vdash \mathbf{L}_1 \Rightarrow \mathbf{L}_2}$

$$\frac{\mathbf{L}' \text{ is a permutation of } \mathbf{L}}{\mathbf{L} \Rightarrow \mathbf{L}'}$$

External lifetime context satisfiability $\boxed{\Gamma \mid \mathbf{E}_1; \mathbf{L}_1 \vdash \mathbf{E}_2}$

$$\mathbf{E}_1; \mathbf{L}_1 \vdash \emptyset \qquad \frac{\mathbf{E}_1; \mathbf{L}_1 \vdash \kappa \sqsubseteq \kappa' \qquad \mathbf{E}_1; \mathbf{L}_1 \vdash \mathbf{E}_2}{\mathbf{E}_1; \mathbf{L}_1 \vdash \mathbf{E}_2, \kappa \sqsubseteq_{\mathrm{e}} \kappa'}$$

1.4.4 Type Inclusion

The main subtyping supported in Rust is lifetime inclusion and (un)folding recursive types. Furthermore, products of unitialized types are equivalent to one large uninitialized type. Finally, type constructors have structural rules witnessing covariance and contravariance of type constructors. This is reflected in our subtyping relation. Some of the rules state an equivalence (⇒), which is meant as sugar for mutual inclusion.

Subtyping $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2}$

T-REFL
$$\mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau$$

T-TRANS
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau' \qquad \mathbf{E}; \mathbf{L} \vdash \tau' \Rightarrow \tau''}{\mathbf{E}; \mathbf{L} \vdash \tau \Rightarrow \tau''}$$

T-BOR-LFT
$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \vdash \&^{\kappa'}_{\mu}\, \tau \Rightarrow \&^{\kappa}_{\mu}\, \tau}$$

T-UNINIT-PROD
$$\mathbf{E}; \mathbf{L} \vdash \lightning_{\Sigma\overline{n}} \Leftrightarrow \Pi\overline{\lightning_{n}}$$

T-REC
$$\frac{\forall \tau_1', \tau_2'.\, (\mathbf{E}; \mathbf{L} \vdash \tau_1' \Rightarrow \tau_2') \Rightarrow (\mathbf{E}; \mathbf{L} \vdash \tau_1[\tau_1'/T_1] \Rightarrow \tau_2[\tau_2'/T_2])}{\mathbf{E}; \mathbf{L} \vdash \mu\, T_1.\, \tau_1 \Rightarrow \mu\, T_2.\, \tau_2}$$

T-REC-UNFOLD
$$\mathbf{E}; \mathbf{L} \vdash \mu\, T.\, \tau \Leftrightarrow \tau[\mu\, T.\, \tau/T]$$

T-OWN
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2}{\mathbf{E}; \mathbf{L} \vdash \mathsf{\textbf{own}}_n\, \tau_1 \Rightarrow \mathsf{\textbf{own}}_n\, \tau_2}$$

T-BOR-SHR
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2}{\mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mathsf{\textbf{shr}}}\, \tau_1 \Rightarrow \&^{\kappa}_{\mathsf{\textbf{shr}}}\, \tau_2}$$

T-BOR-MUT
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau_1 \Leftrightarrow \tau_2}{\mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mathsf{\textbf{mut}}}\, \tau_1 \Leftrightarrow \&^{\kappa}_{\mathsf{\textbf{mut}}}\, \tau_2}$$

T-PROD
$$\frac{\forall i.\, \mathbf{E}; \mathbf{L} \vdash \overline{\tau}_i \Rightarrow \overline{\tau'}_i}{\mathbf{E}; \mathbf{L} \vdash \Pi\overline{\tau} \Rightarrow \Pi\overline{\tau'}}$$

T-SUM
$$\frac{\forall i.\, \mathbf{E}; \mathbf{L} \vdash \overline{\tau}_i \Rightarrow \overline{\tau'}_i}{\mathbf{E}; \mathbf{L} \vdash \Sigma\overline{\tau} \Rightarrow \Sigma\overline{\tau'}}$$

T-FN
$$\frac{\begin{array}{c}\Gamma, \overline{\alpha}', \digamma : \mathsf{\textbf{lft}} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \mathbf{E}[\overline{\kappa}/\overline{\alpha}] \\ \forall i.\, \Gamma, \overline{\alpha}', \digamma : \mathsf{\textbf{lft}} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \overline{\tau}_i' \Rightarrow \overline{\tau}_i \qquad \Gamma, \overline{\alpha}', \digamma : \mathsf{\textbf{lft}} \mid \mathbf{E}', \mathbf{E}_0; \mathbf{L}_0 \vdash \tau \Rightarrow \tau'\end{array}}{\Gamma \mid \mathbf{E}_0; \mathbf{L}_0 \vdash \forall \overline{\alpha}.\, \mathsf{\textbf{fn}}(\digamma : \mathbf{E}; \overline{\tau}) \to \tau \Rightarrow \forall \overline{\alpha}'.\, \mathsf{\textbf{fn}}(\digamma : \mathbf{E}'; \overline{\tau}') \to \tau'}$$

Inclusion of type *contexts* does not just allow applying subtyping; there are also a few coercions supported by the type system. Most notably, a mutable reference can be coerced to a shared reference.

Type context inclusion $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}_1 \Rightarrow \mathbf{T}_2}$

C-PERM
$$\frac{\mathbf{T}' \text{ is a permutation of } \mathbf{T}}{\mathbf{E}; \mathbf{L} \vdash \mathbf{T} \overset{\mathsf{ctx}}{\Rrightarrow} \mathbf{T}'}$$

C-WEAKEN
$$\mathbf{E}; \mathbf{L} \vdash \mathbf{T}, \mathbf{T}' \overset{\mathsf{ctx}}{\Rrightarrow} \mathbf{T}$$

C-FRAME
$$\frac{\mathbf{E}; \mathbf{L} \vdash \mathbf{T}_1 \overset{\mathsf{ctx}}{\Rrightarrow} \mathbf{T}_2}{\mathbf{E}; \mathbf{L} \vdash \mathbf{T}', \mathbf{T}_1 \overset{\mathsf{ctx}}{\Rrightarrow} \mathbf{T}', \mathbf{T}_2}$$

C-COPY
$$\frac{\tau\ \mathsf{copy}}{\mathbf{E}; \mathbf{L} \vdash p \lhd \tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \tau, p \lhd \tau}$$

C-SUBTYPE
$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau \overset{\mathsf{ctx}}{\Rrightarrow} \tau'}{\mathbf{E}; \mathbf{L} \vdash p \lhd \tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \tau'}$$

C-SHARE
$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa\ \mathsf{alive}}{\mathbf{E}; \mathbf{L} \vdash p \lhd \&^{\kappa}_{\mathsf{\textbf{mut}}}\, \tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\kappa}_{\mathsf{\textbf{shr}}}\, \tau}$$

C-SPLIT-OWN
$$\frac{\overline{\tau} \neq [\,] \qquad \forall i.\, m_i = \sum_{j<i} \mathsf{size}(\overline{\tau}_j)}{\mathbf{E}; \mathbf{L} \vdash p \lhd \mathsf{\textbf{own}}_n\, \Pi\overline{\tau} \overset{\mathsf{ctx}}{\Leftrightarrow} \overline{p.m \lhd \mathsf{\textbf{own}}_n\, \tau}}$$

C-SPLIT-BOR
$$\frac{\overline{\tau} \neq [\,] \qquad \forall i.\, m_i = \sum_{j<i} \mathsf{size}(\overline{\tau}_j)}{\mathbf{E}; \mathbf{L} \vdash p \lhd \&^{\kappa}_{\mu}\, \Pi\overline{\tau} \overset{\mathsf{ctx}}{\Leftrightarrow} \overline{p.m \lhd \&^{\kappa}_{\mu}\, \tau}}$$

C-BORROW
$$\mathbf{E}; \mathbf{L} \vdash p \lhd \mathsf{\textbf{own}}_n\, \tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\kappa}_{\mathsf{\textbf{mut}}}\, \tau, p \lhd^{\dagger\kappa} \mathsf{\textbf{own}}_n\, \tau$$

C-REBORROW
$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa' \sqsubseteq \kappa}{\mathbf{E}; \mathbf{L} \vdash p \lhd \&^{\kappa}_{\mu}\, \tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\kappa'}_{\mu}\, \tau, p \lhd^{\dagger\kappa'} \&^{\kappa}_{\mu}\, \tau}$$

The following judgment expresses that when κ ends, we can “unblock” the parts of the typing context that is blocked by κ.

Type context unblocking $\boxed{\Gamma \vdash \mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}$

$$\emptyset \Rightarrow^{\dagger\kappa} \emptyset \qquad \frac{\mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}{\mathbf{T}_1, p \lhd \tau \Rightarrow^{\dagger\kappa} \mathbf{T}_2, p \lhd \tau} \qquad \frac{\mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}{\mathbf{T}_1, p \lhd^{\dagger\kappa} \tau \Rightarrow^{\dagger\kappa} \mathbf{T}_2, p \lhd \tau}$$

$$\frac{\mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}{\mathbf{T}_1, p \lhd^{\dagger\kappa'} \tau \Rightarrow^{\dagger\kappa} \mathbf{T}_2, p \lhd^{\dagger\kappa'} \tau}$$

Continuation context inclusion $\boxed{\Gamma \mid \mathbf{E} \vdash \mathbf{K}_1 \Rightarrow \mathbf{K}_2}$

$$\frac{\mathbf{K}' \text{ is a permutation of } \mathbf{K}}{\mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}'} \qquad \mathbf{E} \vdash \mathbf{K}, \mathbf{K}' \Rightarrow \mathbf{K}$$

$$\frac{\Gamma \mid \mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}' \qquad \Gamma, \overline{x} : \textsf{val} \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T}' \overset{\text{ctx}}{\Rrightarrow} \mathbf{T}}{\Gamma \mid \mathbf{E} \vdash \mathbf{K}, k \lhd \textsf{cont}(\mathbf{L}; \overline{x}.\mathbf{T}) \Rightarrow \mathbf{K}', k \lhd \textsf{cont}(\mathbf{L}; \overline{x}.\mathbf{T}')}$$

1.4.5 Well-typed functions and steps

Finally we come to the main typing judgment: $\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F$ says that F is a well-typed *function body* (as defined by the grammar in §1.3). This means that, under the assumptions described by the contexts, the function is safe to execute. (Functions are in CPS and hence do not return.)

The grammar dictates that a function consists of a bunch of continuations (representing basic blocks) that each consist of a sequence of *instructions*. Instructions *do* return and produce a value, so their typing judgment $\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2$ features two typing contexts: if $\mathbf{T}_1$ holds before the step is executed, then $\mathbf{T}_2$ holds after the step was executed.

We also have two of small helper judgments to express what loading from memory and storing to memory does to types. $\mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_2$ says that we can write something of type τ to a location described by $\tau_!$, which will change the type of the location to τ_2. Similarly, $\mathbf{E}; \mathbf{L} \vdash \tau_1 \multimapinv^{\tau} \tau_2$ says that when reading from a location of type τ_1, we will read something of type τ and the type of the locations changes to τ_2.

Well-typed functions $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F}$

F-consequence

$$\frac{\mathbf{L} \Rightarrow \mathbf{L}' \qquad \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \overset{\text{ctx}}{\Rrightarrow} \mathbf{T}' \qquad \mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}' \qquad \mathbf{E}; \mathbf{L}' \mid \mathbf{K}'; \mathbf{T}' \vdash F}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F}$$

F-equalize

$$\frac{\mathbf{E}, \alpha \sqsubseteq_{\mathrm{e}} \kappa, \kappa \sqsubseteq_{\mathrm{e}} \alpha; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F}{\mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_{\mathrm{l}} [\kappa] \mid \mathbf{K}; \mathbf{T} \vdash F}$$

F-let

$$\frac{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2 \qquad \Gamma, x : \textsf{val} \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}_2, \mathbf{T} \vdash F}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}_1, \mathbf{T} \vdash \mathtt{let}\, x = I \,\mathtt{in}\, F}$$

$$\textsc{F-letcont}\quad \dfrac{\begin{array}{c}\Gamma, k, \overline{x} : \textsf{val} \mid \mathbf{E}; \mathbf{L}_1 \mid \mathbf{K}, k \lhd \textsf{cont}(\mathbf{L}_1; \overline{x}.\, \mathbf{T}'); \mathbf{T}' \vdash F_1 \\ \Gamma, k : \textsf{val} \mid \mathbf{E}; \mathbf{L}_2 \mid \mathbf{K}, k \lhd \textsf{cont}(\mathbf{L}_1; \overline{x}.\, \mathbf{T}'); \mathbf{T} \vdash F_2\end{array}}{\Gamma \mid \mathbf{E}; \mathbf{L}_2 \mid \mathbf{K}; \mathbf{T} \vdash \texttt{letcont}\, k(\overline{x}) := F_1 \,\texttt{in}\, F_2}$$

$$\textsc{F-if}\quad \dfrac{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F_1 \qquad \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash F_2}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \textsf{bool} \vdash \texttt{if}\, p \,\texttt{then}\, F_1 \,\texttt{else}\, F_2}$$

$$\textsc{F-jump}\quad \dfrac{\mathbf{E}; \mathbf{L} \vdash \mathbf{T} \Rightarrow \mathbf{T}'[\overline{y}/\overline{x}]}{\mathbf{E}; \mathbf{L} \mid k \lhd \textsf{cont}(\mathbf{L}; \overline{x}.\, \mathbf{T}'); \mathbf{T} \vdash \texttt{jump}\, k(\overline{y})}$$

$$\textsc{F-call}\quad \dfrac{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \mathbf{T} \Rightarrow \overline{p} \mathrel{\overline{\lhd}} \textsf{own}\, \overline{\tau}, \mathbf{T}' \qquad \mathbf{E}; \mathbf{L} \vdash \overline{\kappa} \;\textsf{alive} \qquad \Gamma, \digamma : \textsf{lft} \mid \mathbf{E}, \digamma \sqsubseteq_{\mathrm{e}} \overline{\kappa}; \mathbf{L} \vdash \mathbf{E}'}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid k \lhd \textsf{cont}(\mathbf{L}; y.\, y \lhd \textsf{own}\, \tau, \mathbf{T}'); \mathbf{T}, f \lhd \textsf{fn}(\digamma : \mathbf{E}'; \overline{\tau}) \to \tau \vdash \texttt{call}\, f(\overline{p}) \,\texttt{ret}\, k}$$

$$\textsc{F-newlft}\quad \dfrac{\Gamma, \alpha : \textsf{lft} \mid \mathbf{E}; \mathbf{L}, \alpha \sqsubseteq_{\mathrm{l}} \overline{\kappa} \mid \mathbf{K}; \mathbf{T} \vdash F}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T} \vdash \texttt{newlft}; F}$$

$$\textsc{F-endlft}\quad \dfrac{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}' \vdash F \qquad \mathbf{T} \Rightarrow^{\dagger\kappa} \mathbf{T}'}{\mathbf{E}; \mathbf{L}, \kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} \mid \mathbf{K}; \mathbf{T} \vdash \texttt{endlft}; F}$$

$$\textsc{F-case-own}\quad \dfrac{\begin{array}{c}\forall i.\, (\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p.0 \lhd \textsf{own}_n \lightning, p.1 \lhd \textsf{own}_n\, \overline{\tau}_i, p.(1 + \textsf{size}(\overline{\tau}_i)) \lhd \textsf{own}_n \lightning_{(\max_j \textsf{size}(\overline{\tau}_j)) - \textsf{size}(\overline{\tau}_i)} \vdash F_i) \vee \\ (\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \textsf{own}_n\, \Sigma\overline{\tau} \vdash F_i)\end{array}}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \textsf{own}_n\, \Sigma\overline{\tau} \vdash \texttt{case}\, {}^{*}p \,\texttt{of}\, \overline{F}}$$

$$\textsc{F-case-bor}\quad \dfrac{\mathbf{E}; \mathbf{L} \vdash \kappa \;\textsf{alive} \qquad \forall i.\, (\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p.1 \lhd \&^{\kappa}_{\mu}\, \tau_i \vdash F_i) \vee (\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \&^{\kappa}_{\mu}\, \Sigma\overline{\tau} \vdash F_i)}{\mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \&^{\kappa}_{\mu}\, \Sigma\overline{\tau} \vdash \texttt{case}\, {}^{*}p \,\texttt{of}\, \overline{F}}$$

Type writing $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_2}$

$$\textsc{Twrite-own}\quad \dfrac{\textsf{size}(\tau) = \textsf{size}(\tau')}{\mathbf{E}; \mathbf{L} \vdash \textsf{own}_n\, \tau' \multimap^{\tau} \textsf{own}_n\, \tau} \qquad \textsc{Twrite-bor}\quad \dfrac{\mathbf{E}; \mathbf{L} \vdash \kappa \;\textsf{alive}}{\mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\textsf{mut}}\, \tau \multimap^{\tau} \&^{\kappa}_{\textsf{mut}}\, \tau}$$

Type reading $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimapinv^{\tau} \tau_2}$

$$\textsc{Tread-own-copy}\quad \dfrac{\tau \;\textsf{copy}}{\mathbf{E}; \mathbf{L} \vdash \textsf{own}_n\, \tau \multimapinv^{\tau} \textsf{own}_n\, \tau} \qquad \textsc{Tread-own-move}\quad \dfrac{n = \textsf{size}(\tau)}{\mathbf{E}; \mathbf{L} \vdash \textsf{own}_m\, \tau \multimapinv^{\tau} \textsf{own}_m \lightning_n} \qquad \textsc{Tread-bor}\quad \dfrac{\tau \;\textsf{copy} \qquad \mathbf{E}; \mathbf{L} \vdash \kappa \;\textsf{alive}}{\mathbf{E}; \mathbf{L} \vdash \&^{\kappa}_{\mu}\, \tau \multimapinv^{\tau} \&^{\kappa}_{\mu}\, \tau}$$

Well-typed instructions $\boxed{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2}$

$$\textsc{S-true}\quad \mathbf{E}; \mathbf{L} \mid \emptyset \vdash \texttt{true} \dashv x.\, x \lhd \textsf{bool} \qquad \textsc{S-false}\quad \mathbf{E}; \mathbf{L} \mid \emptyset \vdash \texttt{false} \dashv x.\, x \lhd \textsf{bool} \qquad \textsc{S-num}\quad \mathbf{E}; \mathbf{L} \mid \emptyset \vdash z \dashv x.\, x \lhd \textsf{int}$$

S-FN

$$\frac{\overline{\tau}'\ \mathsf{copy} \qquad \overline{\tau}'\ \mathsf{send} \qquad \begin{array}{r}\Gamma, \overline{\alpha}, \digamma : \textbf{lft}, f, \overline{x}, k : \textbf{val} \mid \mathbf{E}, \mathbf{E}'; \digamma \sqsubseteq_{\mathrm{l}} [] \mid k \lhd \textbf{cont}(\digamma \sqsubseteq_{\mathrm{l}} []; y.\, y \lhd \textbf{own}\, \tau);\\ \overline{p}\ \overline{\lhd}\ \overline{\tau}', \overline{x}\ \overline{\lhd}\ \textbf{own}\, \overline{\tau}, f \lhd \forall \overline{\alpha}.\, \textbf{fn}(\digamma : \mathbf{E}; \overline{\tau}) \to \tau \vdash F\end{array}}{\Gamma \mid \mathbf{E}'; \mathbf{L}' \mid \overline{p}\ \overline{\lhd}\ \overline{\tau}' \vdash \texttt{funrec}\, f(\overline{x})\, \texttt{ret}\, k := F \dashv f.\, f \lhd \forall \overline{\alpha}.\, \textbf{fn}(\digamma : \mathbf{E}; \overline{\tau}) \to \tau}$$

S-PATH

$$\mathbf{E}; \mathbf{L} \mid p \lhd \tau \vdash p \dashv x.\, x \lhd \tau$$

S-NAT-OP

$$\mathbf{E}; \mathbf{L} \mid p_1 \lhd \textbf{int}, p_2 \lhd \textbf{int} \vdash p_1 \{+, -\}\, p_2 \dashv x.\, x \lhd \textbf{int}$$

S-NAT-LEQ

$$\mathbf{E}; \mathbf{L} \mid p_1 \lhd \textbf{int}, p_2 \lhd \textbf{int} \vdash p_1 \leq p_2 \dashv x.\, x \lhd \textbf{bool}$$

S-NEW

$$\mathbf{E}; \mathbf{L} \mid \emptyset \vdash \texttt{new}(n) \dashv x.\, x \lhd \textbf{own}_n \lightning_n$$

S-DELETE

$$\frac{n = \mathsf{size}(\tau)}{\mathbf{E}; \mathbf{L} \mid p \lhd \textbf{own}_n\, \tau \vdash \texttt{delete}(n, p) \dashv \emptyset}$$

S-DEREF

$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau_1 \multimapinv^{\tau} \tau_1' \qquad \mathsf{size}(\tau) = 1}{\mathbf{E}; \mathbf{L} \mid p \lhd \tau_1 \vdash {}^*p \dashv x.\, p \lhd \tau_1', x \lhd \tau}$$

S-DEREF-BOR-OWN

$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa\ \mathsf{alive}}{\mathbf{E}; \mathbf{L} \mid p \lhd \&_\mu^\kappa\, \textbf{own}_n\, \tau \vdash {}^*p \dashv x.\, x \lhd \&_\mu^\kappa\, \tau}$$

S-DEREF-BOR-BOR

$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa\ \mathsf{alive} \qquad \mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}{\mathbf{E}; \mathbf{L} \mid p \lhd \&_\mu^\kappa\, \&_{\textbf{mut}}^{\kappa'}\, \tau \vdash {}^*p \dashv x.\, x \lhd \&_\mu^\kappa\, \tau}$$

S-ASSGN

$$\frac{\mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_1'}{\mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau \vdash p_1 := p_2 \dashv p_1 \lhd \tau_1'}$$

S-SUM-ASSGN-UNIT

$$\frac{\overline{\tau}_i = \Pi[\,] \qquad \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\Sigma \overline{\tau}} \tau_1'}{\mathbf{E}; \mathbf{L} \mid p \lhd \tau_1 \vdash p \overset{\mathsf{inj}\, i}{:==} () \dashv p \lhd \tau_1'}$$

S-SUM-ASSGN

$$\frac{\overline{\tau}_i = \tau \qquad \tau_1 \multimap^{\Sigma \overline{\tau}} \tau_1'}{\mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau \vdash p_1 \overset{\mathsf{inj}\, i}{:==} p_2 \dashv p_1 \lhd \tau_1'}$$

S-MEMCPY

$$\frac{\mathsf{size}(\tau) = n \qquad \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_1' \qquad \mathbf{E}; \mathbf{L} \vdash \tau_2 \multimapinv^{\tau} \tau_2'}{\mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau_2 \vdash p_1 :=_n {}^*p_2 \dashv p_1 \lhd \tau_1', p_2 \lhd \tau_2'}$$

S-SUM-MEMCPY

$$\frac{\mathsf{size}(\tau) = n \qquad \mathbf{E}; \mathbf{L} \vdash \tau_1 \multimap^{\Sigma \overline{\tau}} \tau_1' \qquad \mathbf{E}; \mathbf{L} \vdash \tau_2 \multimapinv^{\tau} \tau_2' \qquad \overline{\tau}_i = \tau}{\mathbf{E}; \mathbf{L} \mid p_1 \lhd \tau_1, p_2 \lhd \tau_2 \vdash p_1 \overset{\mathsf{inj}\, i}{:==}_n {}^*p_2 \dashv p_1 \lhd \tau_1', p_2 \lhd \tau_2'}$$

2 Some examples

This section contains some manually type-checked functions demonstrating how the type system looks like in action.

We write $\tau_1 \times \tau_2 \times \ldots$ for $\Pi\overline{\tau}$, $\mathbf{()}$ for $\Pi[\,]$, $\tau_1 + \tau_2 + \ldots$ for $\Sigma\overline{\tau}$ and $\mathbf{!}$ for $\Sigma[\,]$. We use $\lightning$ as sugar for $\lightning_1$. Finally, $\mathsf{\textbf{own}}\,\tau$ is short for $\mathsf{\textbf{own}}_{\mathsf{size}(\tau)}\,\tau$.

In local lifetime contexts, we use α as notation for $\alpha \sqsubseteq_{\mathrm{l}} []$.

It turns out to be useful to have some syntactic sugar for calling a function and using its return value, for declaring continuations without writing code "backwards", and for immediately initializing a fresh allocation.

$$\begin{aligned}
\texttt{havecont}\,k\,\texttt{in}\,F_1\,\texttt{wherecont}\,k(\overline{x}) := F_2 \;&:=\; \texttt{letcont}\,k(x) := F_2\,\texttt{in}\,F_1 \\
\texttt{letcall}\,x = f(\overline{p})\,\texttt{in}\,F \;&:=\; \texttt{letcont}\,k(x) := F\,\texttt{in}\,\texttt{call}\,f(\overline{p})\,\texttt{ret}\,k \\
\texttt{letalloc}\,x : \tau := p\,\texttt{in}\,F \;&:=\; \texttt{let}\,x = \texttt{new}(1)\,\texttt{in}\,x := p\,\texttt{in}\,F \\
\texttt{letalloc}\,x : \tau := {}^{*}p\,\texttt{in}\,F \;&:=\; \texttt{let}\,x = \texttt{new}(\tau)\,\texttt{in}\,x :=_{\tau} {}^{*}p\,\texttt{in}\,F
\end{aligned}$$

Notice that we sometimes use types as subscripts where the syntax expects a number. In this case, we implicitly refer to the size of the type.

The syntactic sugar above enjoys the typing rules below.

F-letcall

$$\frac{\Gamma, x : \mathsf{\textbf{val}} \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, x \lhd \tau \vdash F \qquad \mathbf{E}; \mathbf{L} \vdash \mathbf{E}'}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, f \lhd \mathsf{\textbf{fn}}(\mathbf{E}'; \overline{\tau}) \to \tau, \overline{p} \lhd \overline{\tau} \vdash \texttt{letcall}\,x = f(\overline{p})\,\texttt{in}\,F}$$

F-letalloc-assgn

$$\frac{\Gamma, x : \mathsf{\textbf{val}} \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, x \lhd \mathsf{\textbf{own}}\,\tau \vdash F}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \tau \vdash \texttt{letalloc}\,x : \tau := p\,\texttt{in}\,F}$$

F-letalloc-memcpy

$$\frac{\tau_1 \multimapinv^{\tau} \tau_2 \qquad \Gamma, x : \mathsf{\textbf{val}} \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, x \lhd \mathsf{\textbf{own}}\,\tau, p \lhd \tau_2 \vdash F}{\Gamma \mid \mathbf{E}; \mathbf{L} \mid \mathbf{K}; \mathbf{T}, p \lhd \tau_1 \vdash \texttt{letalloc}\,x : \tau := {}^{*}p\,\texttt{in}\,F}$$

Example 1: Mutable to shared reference, field reference.

```
struct Point { x: i32, y: i32 }
fn get_x<'a>(p: &'a mut Point) -> &'a i32 {
    &(*p).x
}
```

The types translate as follows:

$$\begin{aligned}
&\mathit{Point} := \mathsf{\textbf{int}} \times \mathsf{\textbf{int}} \\
&\mathit{get\_x} := \forall\alpha.\,\mathsf{\textbf{fn}}(\digamma : \digamma \sqsubseteq_{\mathrm{e}} \alpha; \&^{\alpha}_{\mathsf{\textbf{mut}}}\,\mathit{Point}) \to \&^{\alpha}_{\mathsf{\textbf{shr}}}\,\mathsf{\textbf{int}}
\end{aligned}$$

All lifetime bounds (in particular, the fact that lifetime parameters are alive) all have to be made explicit. Furthermore, all variables and return values are passed via owned pointers.

The code itself translates to:

$$\begin{aligned}
&\texttt{funrec}\,\mathtt{\mathit{get\_x}}(\mathtt{\mathit{p}})\,\texttt{ret}\,\mathtt{\mathit{ret}} := \\
&\quad \texttt{let}\,\mathtt{\mathit{p'}} = *\mathtt{\mathit{p}}\,\texttt{in}\,\texttt{letalloc}\,\mathtt{\mathit{r}} : \&^{\alpha}_{\mathsf{\textbf{shr}}}\,\mathsf{\textbf{int}} := \mathtt{\mathit{p'}}.0\,\texttt{in} \\
&\quad \texttt{delete}(\&^{\alpha}_{\mathsf{\textbf{mut}}}\,\mathit{Point}, \mathtt{\mathit{p}}); \texttt{jump}\,\mathtt{\mathit{ret}}(\mathtt{\mathit{r}})
\end{aligned}$$

I will usually try to make the Rust code and the $\lambda_{\textsf{Rust}}$ code match up in terms of lines, so one line of code in the original function corresponds to one line of code in the translation. At the end, there will always be an additional line deallocating the stack frame and jumping to the return continuation.

Now we can typecheck the function body.

Context: $\digamma \sqsubseteq_1 \alpha$

$\{\mathit{ret} \lhd \mathbf{cont}(\digamma; r.\, r \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{shr}}\ \mathbf{int}); p \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{mut}}\ \mathit{Point}\}$

$\quad \{p\}\ \mathtt{let}\ p' = *p\ \mathtt{in}\ \{p' \lhd \&^{\alpha}_{\mathbf{mut}}\ \mathit{Point}, p \lhd \mathbf{own}\ ↯\}$

$\quad \{p'\}\ \text{Split}\ \{p'.0 \lhd \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}, p'.1 \lhd \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}\}$

$\quad \{p'.0\}\ \mathtt{letalloc}\ r : \&^{\alpha}_{\mathbf{shr}}\ \mathbf{int} := p'.0\ \mathtt{in}\ \{r \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{shr}}\ \mathbf{int}\}$

$\quad \{p, \mathit{ret}, r\}\ \mathtt{delete}(\&^{\alpha}_{\mathbf{mut}}\ \mathit{Point}, p); \mathit{ret}(r)$

A quick note on our typing outline conventions: In the very first line, we state the external lifetime context E, which does not change during verification. The line below that state the remaining contexts. The next lines are indented, which means that they describe a *change* in the contexts – the items on the left are used, the items on the right are produced. Formally, such lines correspond to the application of one of the rules for well-typed programs. Used items will, in general, be considered gone because our contexts are substructural. However, if the item permits duplication (*e.g.*, for $x \lhd \tau$ for τ $\textsf{copy}$), the item will implicitly be duplicated. When specifying items of the form to be consumed, we will often shorten $X \lhd \_$ to just X. This still uniquely identifies the used-up context item.

This choice of notation prevents repeating the same items over and over; however, the current context is fairly implicit: the context consists of all the items that are produced by any preceeding step, minus consumed non-duplicable items. When appropriate, we will repeat the entire current context in an un-indented line to keep things clearer.

Example 2: Copying a reference out of ownership

```
fn rebor(mut t1: Point, mut t2: Point) -> i32 {
    let mut x = &mut t1;
    let y = &mut (*{x}).y;
    x = &mut t2;
    *y
}
```

In the code above, dereferencing `x` *consumes* that pointer, so its permission is gone – it is now essentially uninitialized. This is why `x` is wrapped in curly braces: if we had just written *(*x).y*, Rust would instead have *re-borrowed* `x` so the program would not typecheck.

$$\mathit{rebor} := \mathbf{fn}(\mathit{Point}, \mathit{Point}) \to \mathbf{int}$$

```
funrec $\mathit{rebor}(t1, t2)$ ret $\mathit{ret}$ :=
  newlft;
  letalloc $x : \&^{\alpha}_{\mathsf{mut}}\ \mathit{Point} := t1$;
  let $x' = {}^{*}x$ in let $y = x'.0$ in
  $x := t2$ in
  let $y' = {}^{*}y$ in letalloc $r : \mathsf{int} := y'$ in
  endlft; delete($\mathit{Point}, t1$); delete($\mathit{Point}, t2$); delete($\&^{\alpha}_{\mathsf{mut}}\ \mathit{Point}, x$); jump $\mathit{ret}(r)$
```

```
$\{\mathit{ret} \lhd \mathsf{cont}(\digamma; r.\ r \lhd \mathsf{own\ int}); t1 \lhd \mathsf{own}\ \mathit{Point}, t2 \lhd \mathsf{own}\ \mathit{Point}\}$
  newlft;
Local lifetimes: $\alpha \sqsubseteq_{\mathrm{l}} [\,]$
  $\{t1, t2\}$ Borrow at $\alpha$ $\{t1 \lhd \&^{\alpha}_{\mathsf{mut}}\ \mathit{Point}, t1 \lhd^{\dagger\alpha} \mathsf{own}\ \mathit{Point}, t2 \lhd \&^{\alpha}_{\mathsf{mut}}\ \mathit{Point}, t2 \lhd^{\dagger\alpha} \mathsf{own}\ \mathit{Point}\}$
  $\{t1\}$ letalloc $x : \&^{\alpha}_{\mathsf{mut}}\ \mathit{Point} := t1$; $\{x \lhd \mathsf{own}\ \&^{\alpha}_{\mathsf{mut}}\ \mathit{Point}\}$
  $\{x\}$ let $x' = {}^{*}x$ in $\{x' \lhd \&^{\alpha}_{\mathsf{mut}}\ \mathit{Point}, x \lhd \mathsf{own}\ \lightning\}$
  $\{x'\}$ let $y = x'.0$ in $\{y \lhd \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}\}$
  $\{x, t2\}$ $x := t2$ in $\{x \lhd \mathsf{own}\ \&^{\alpha}_{\mathsf{mut}}\ \mathit{Point}\}$
  $\{y\}$ let $y' = {}^{*}y$ in $\{y' \lhd \mathsf{int}, y\}$
  $\{y'\}$ letalloc $r : \mathsf{int} := y'$ $\{r \lhd \mathsf{own\ int}\}$
  $\{t1, t2\}$ endlft; $\{t1 \lhd \mathsf{own}\ \mathit{Point}, t2 \lhd \mathsf{own}\ \mathit{Point}\}$
Local lifetimes: $\emptyset$
  $\{t1, t2, x, \mathit{ret}, r\}$ delete($\mathit{Point}, t1$); delete($\mathit{Point}, t2$); delete($\&^{\alpha}_{\mathsf{mut}}\ \mathit{Point}, x$); jump $\mathit{ret}(r)$
```

Example 3: Borrowing from a borrowed box.

Now we can consider this piece of Rust code:

```rust
fn unbox<'a>(b: &'a mut Box<Point>) -> &'a mut u32 {
  let bx = &mut *b;
  &mut (*bx).x
}
```

Rust boxes translate to owned pointers. In Rust, the only semantic difference between the type T and Box<T> is that the former lives on the stack, while the latter lives on the heap. λ_{Rust} does not distinguish between stack and heap, so the two concepts unify: Box is just **own** .

$$\mathit{unbox} := \forall\alpha.\ \mathsf{fn}(\digamma : \digamma \sqsubseteq_{\mathrm{e}} \alpha; \&^{\alpha}_{\mathsf{mut}}\ \mathsf{own}\ \mathit{Point}) \to \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int}$$

```
funrec $\mathit{unbox}(b)$ ret $\mathit{ret}$ :=
  let $b' = {}^{*}b$ in let $bx = {}^{*}b'$ in
  letalloc $r : \&^{\alpha}_{\mathsf{mut}}\ \mathsf{int} := bx.0$ in
  delete($\&^{\alpha}_{\mathsf{mut}}\ \mathit{Point}, b$); delete($\mathit{Point}, bx$); jump $\mathit{ret}(r)$
```

Context: $\digamma \sqsubseteq_1 \alpha$

$\{\mathtt{ret} \lhd \mathbf{cont}(\digamma; \mathtt{r}.\, \mathtt{r} \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int});\ \mathtt{b} \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{mut}}\ \mathbf{own}\ \mathit{Point}\}$
$\quad\{\mathtt{b}\}\ \mathtt{let}\ \mathtt{b'} = {}^{*}\mathtt{b}\ \mathtt{in}\ \ \{b' \lhd \&^{\alpha}_{\mathbf{mut}}\ \mathbf{own}\ \mathit{Point}, \mathtt{b} \lhd \mathbf{own}\ \lightning\}$
$\quad\{\mathtt{b'}\}\ \mathtt{let}\ \mathtt{bx} = {}^{*}\mathtt{b'}\ \mathtt{in}\ \ \{\mathtt{bx} \lhd \mathbf{own}\ \mathit{Point}\}$
$\quad\{\mathtt{bx}\}$ Split $\{\mathtt{bx}.0 \lhd \mathbf{own}_{\mathit{Point}}\ \mathbf{int}, \mathtt{bx}.1 \lhd \mathbf{own}_{\mathit{Point}}\ \mathbf{int}\}$
$\quad\{\mathtt{bx}.0\}\ \mathtt{letalloc}\ \mathtt{r} : \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int} := \mathtt{bx}.0\ \mathtt{in}\ \ \{\mathtt{r} \lhd \mathbf{own}\ \&^{\alpha}_{\mathbf{mut}}\ \mathbf{int}, \mathtt{bx}.0\}$
$\quad\{\mathtt{bx}.0, \mathtt{bx}.1\}$ Merge $\{\mathtt{bx} \lhd \mathbf{own}\ \mathit{Point}\}$
$\quad\{\mathtt{b}, \mathtt{bx}, \mathtt{r}, \mathtt{retval}\}\ \mathtt{delete}(\&^{\alpha}_{\mathbf{mut}}\ \mathit{Point}, \mathtt{b}); \mathtt{delete}(\mathit{Point}, \mathtt{bx}); \mathtt{jump}\ \mathtt{ret}(\mathtt{r})$

Example 4: Struct initialization.

```rust
fn point(x: i32, y: i32) -> Point {
    Point { x: x, y: y}
}
```

$$\mathit{point} := \mathbf{fn}(\mathbf{int}, \mathbf{int}) \to \mathit{Point}$$

$$\begin{array}{l}
\mathtt{funrec}\ \mathtt{point}(\mathtt{x}, \mathtt{y})\ \mathtt{ret}\ \mathtt{ret} := \\
\quad \mathtt{let}\ \mathtt{x'} = {}^{*}\mathtt{x}\ \mathtt{in}\ \mathtt{let}\ \mathtt{y'} = {}^{*}\mathtt{y}\ \mathtt{in} \\
\quad \mathtt{let}\ \mathtt{r} = \mathtt{new}(\mathit{Point})\ \mathtt{in} \\
\quad \mathtt{r}.0 := x'; \mathtt{r}.1 := y'; \\
\quad \mathtt{delete}(\mathbf{int}, \mathtt{x}); \mathtt{delete}(\mathbf{int}, \mathtt{y}); \mathtt{jump}\ \mathtt{ret}(\mathtt{r})
\end{array}$$

The part about having the lines match up does not really work out here any more...

$\{\mathtt{ret} \lhd \mathbf{cont}(\digamma; \mathtt{r}.\, \mathtt{r} \lhd \mathbf{own}\ \mathit{Point}); \mathtt{x} \lhd \mathbf{own}\ \mathbf{int}, \mathtt{y} \lhd \mathbf{own}\ \mathbf{int}\}$
$\quad\{\mathtt{x}, \mathtt{y}\}\ \mathtt{let}\ \mathtt{x'} = {}^{*}\mathtt{x}\ \mathtt{in}\ \mathtt{let}\ \mathtt{y'} = {}^{*}\mathtt{y}\ \mathtt{in}\ \ \{\mathtt{x} \lhd \mathbf{own}\ \lightning, \mathtt{x'} \lhd \mathbf{int}, \mathtt{y} \lhd \mathbf{own}\ \lightning, \mathtt{y'} \lhd \mathbf{int}\}$
$\quad\{\}\ \mathtt{let}\ \mathtt{r} = \mathtt{new}(\mathit{Point})\ \mathtt{in}\ \ \{\mathtt{r} \lhd \mathbf{own}\ \lightning_{\mathit{Point}}\}$
$\quad\{\mathtt{r}\}$ Split $\{\mathtt{r}.0 \lhd \mathbf{own}_{\mathit{Point}}\ \lightning, \mathtt{r}.1 \lhd \mathbf{own}_{\mathit{Point}}\ \lightning\}$
$\quad\{\mathtt{r}.0, \mathtt{x'}\}\ \mathtt{r}.0 := x';\ \{\mathtt{r}.0 \lhd \mathbf{own}_{\mathit{Point}}\ \mathbf{int}, \mathtt{x'}\}$
$\quad\{\mathtt{r}.1, \mathtt{y'}\}\ \mathtt{r}.1 := y';\ \{\mathtt{r}.1 \lhd \mathbf{own}_{\mathit{Point}}\ \mathbf{int}, \mathtt{y'}\}$
$\quad\{\mathtt{r}.0, \mathtt{r}.1\}$ Merge $\{\mathtt{r} \lhd \mathbf{own}\ \mathit{Point}\}$
$\quad\{\mathtt{x}, \mathtt{y}, \mathtt{ret}, \mathtt{r}\}\ \mathtt{delete}(\mathbf{int}, \mathtt{x}); \mathtt{delete}(\mathbf{int}, \mathtt{y}); \mathtt{jump}\ \mathtt{ret}(\mathtt{r})$

Example 5: Enum matching and initialization.

We assume the following *meta-level* type definition:

$$\mathit{Option} := \lambda\tau.\, \mathbf{()} + \tau$$

We will write $\mathit{Option}\langle\tau\rangle$ for τ applied to Option.

```rust
fn option_as_mut<T>(o: &mut Option<T>) -> Option<&mut T> {
    match *o {
        None => None,
        Some(ref mut t) => Some(t)
    }
}
```

option_as_mut is parametric over some type τ on the meta-level.

$$option\_as\_mut\langle\tau\rangle := \forall\alpha.\, \mathbf{fn}(ϝ : ϝ \sqsubseteq_{\mathrm{e}} \alpha; \&^{\alpha}_{\mathsf{mut}}\, Option\langle\tau\rangle) \to Option\langle\&^{\alpha}_{\mathsf{mut}}\, \tau\rangle$$

```
funrec $option\_as\_mut(o)$ ret $ret$ :=
  let $r$ = new($Option\langle\&^{\alpha}_{\mathsf{mut}}\,\tau\rangle$) in
havecont $k$ in
  let $o'$ = $^{*}o$ in case $^{*}o'$ of
  – $r \overset{\mathsf{inj}\,0}{:==} ()$;
    jump $k()$
  – $r \overset{\mathsf{inj}\,1}{:==} o'.1$;
    jump $k()$
wherecont $k()$ :=
  delete($\mathsf{own}\,\&^{\alpha}_{\mathsf{mut}}\, Option\langle\tau\rangle$, $o$); jump $ret(r)$
```

```
Context: $ϝ \sqsubseteq_{1} \alpha$
$\{ret \lhd \mathsf{cont}(ϝ; r.\, r \lhd \mathsf{own}\, Option\langle\&^{\alpha}_{\mathsf{mut}}\,\tau\rangle); o \lhd \mathsf{own}\,\&^{\alpha}_{\mathsf{mut}}\, Option\langle\tau\rangle\}$
  $\{\}$ let $r$ = new($Option\langle\&^{\alpha}_{\mathsf{mut}}\,\tau\rangle$) in $\{r \lhd \mathsf{own} \lightning_{Option\langle\&^{\alpha}_{\mathsf{mut}}\,\tau\rangle}\}$
havecont $k$ in
$\{k \lhd \mathsf{cont}(ϝ; r \lhd \mathsf{own}\, Option\langle\&^{\alpha}_{\mathsf{mut}}\,\tau\rangle, o \lhd \mathsf{own} \lightning); o \lhd \mathsf{own}\,\&^{\alpha}_{\mathsf{mut}}\, Option\langle\tau\rangle, r \lhd \mathsf{own} \lightning_{Option\langle\&^{\alpha}_{\mathsf{mut}}\,\tau\rangle}\}$
  $\{o\}$ let $o'$ = $^{*}o$ in $\{o' \lhd \&^{\alpha}_{\mathsf{mut}}\, Option\langle\tau\rangle, o \lhd \mathsf{own} \lightning\}$
  $\{o'\}$ case $^{*}o'$ of
  – $\{o'.1 \lhd \&^{\alpha}_{\mathsf{mut}}\, ()\}$
    $\{r\}$ $r \overset{\mathsf{inj}\,0}{:==} ()$ in $\{r \lhd \mathsf{own}\, Option\langle\&^{\alpha}_{\mathsf{mut}}\,\tau\rangle\}$
    $\{k, o, r\}$ jump $k()$
  – $\{o'.1 \lhd \&^{\alpha}_{\mathsf{mut}}\, \tau\}$
    $\{o'.1\}$ $r \overset{\mathsf{inj}\,1}{:==} o'.1$ in $\{r \lhd \mathsf{own}\, Option\langle\&^{\alpha}_{\mathsf{mut}}\,\tau\rangle\}$
    $\{k, o, r\}$ jump $k()$
wherecont $k()$ :=
$\{r \lhd \mathsf{own}\, Option\langle\&^{\alpha}_{\mathsf{mut}}\,\tau\rangle, o \lhd \mathsf{own} \lightning\}$
  $\{o, ret, r\}$ delete($\mathsf{own}\,\&^{\alpha}_{\mathsf{mut}}\, Option\langle\tau\rangle$, $o$); jump $ret(r)$
```

Example 6: Moving out of an enum.

```rust
fn unwrap_or<T>(o: Option<T>, def: T) -> T {
  match o {
    None => def,
    Some(t) => t
  }
}
```

unwrap_or is parametric over some type τ on the meta-level.

$$unwrap\_or := \forall\alpha.\, \mathbf{fn}(ϝ : ϝ \sqsubseteq_{\mathrm{e}} \alpha; Option\langle\tau\rangle, \tau) \to \tau$$

```
$\mathtt{funrec}\ \mathit{unwrap\_or}(o, \mathit{def})\ \mathtt{ret}\ \mathit{ret} :=$
  $\mathtt{case}\ {}^{*}o\ \mathtt{of}$
  $-\ \mathtt{delete}(\mathit{Option}\langle\tau\rangle, o); \mathtt{jump}\ \mathit{ret}(\mathit{def})$
  $-\ \mathtt{letalloc}\ r : \tau := o.1\ \mathtt{in}$
    $\mathtt{delete}(\mathit{Option}\langle\tau\rangle, o); \mathtt{delete}(\tau, \mathit{def}); \mathtt{jump}\ \mathit{ret}(r)$
```

```
$\{\mathit{ret} \lhd \mathbf{cont}(\mathrm{F}; r.\ r \lhd \mathbf{own}\ \tau); o \lhd \mathbf{own}\ \mathit{Option}\langle\tau\rangle, \mathit{def} \lhd \mathbf{own}\ \tau\}$
  $\{o\}\ \mathtt{case}\ {}^{*}o\ \mathtt{of}$
  $-\ \{o \lhd \mathbf{own}\ \mathit{Option}\langle\tau\rangle\}$
    $\{o, \mathit{ret}, \mathit{def}\}\ \mathtt{delete}(\mathit{Option}\langle\tau\rangle, o); \mathtt{jump}\ \mathit{ret}(\mathit{def})$
  $-\ \{o.0 \lhd \mathbf{own}_{\mathit{Option}\langle\tau\rangle}\ \lightning, o.1 \lhd \mathbf{own}_{\mathit{Option}\langle\tau\rangle}\ \tau\}$
    $\{o.1\}\ \mathtt{letalloc}\ r : \tau := o.1\ \mathtt{in}\ \{r \lhd \mathbf{own}\ \tau, o.1 \lhd \mathbf{own}_{\mathit{Option}\langle\tau\rangle}\ \lightning_{\tau}\}$
    $\{o.1, o.1\}$ Merge $\{o \lhd \mathbf{own}\ \lightning_{\mathit{Option}\langle\tau\rangle}\}$
    $\{o, \mathit{def}, \mathit{ret}, r\}\ \mathtt{delete}(\mathit{Option}\langle\tau\rangle, o); \mathtt{delete}(\tau, \mathit{def}); \mathtt{jump}\ \mathit{ret}(r)$
```

Example 7: Lazy lifetime initialization.

```
struct Two<'a> {
    f: &'a i32,
    g: &'a i32,
}

fn lazy_lft() {
    let (mut t, f, g) : (Two, i32, i32);
    f = 42;
    t = Two { f: &f, g: &f };
    *t.f; // The lifetime definitely is already active here
    g = 23; // And g is definitely not yet borrowed.
    t.g = &g; // But now we can borrow g at the *old* lifetime.
}
```

Let $\mathit{Two}\langle\kappa\rangle := \&^{\kappa}_{\mathsf{shr}}\ \mathbf{int} \times \&^{\kappa}_{\mathsf{shr}}\ \mathbf{int}$.

$$\mathit{lazy\_lft} := \mathbf{fn}() \to \mathbf{()}$$

```
$\mathtt{funrec}\ \mathit{lazy\_lft}()\ \mathtt{ret}\ \mathit{ret} :=$
  $\mathtt{newlft};$
  $\mathtt{let}\ t = \mathtt{new}(2)\ \mathtt{in}\ \mathtt{let}\ f = \mathtt{new}(1)\ \mathtt{in}\ \mathtt{let}\ g = \mathtt{new}(1)\ \mathtt{in}$
  $f := 42;$
  $t.0 := f; t.1 := f;$
  $\mathtt{let}\ \mathit{t0'} = {}^{*}t.0\ \mathtt{in}\ {}^{*}\mathit{t0'};$
  $g := 23;$
  $t.1 := g;$
  $\mathtt{let}\ r = \mathtt{new}(0)\ \mathtt{in}$
  $\mathtt{endlft}; \mathtt{delete}(2, t); \mathtt{delete}(1, f); \mathtt{delete}(1, g); \mathtt{jump}\ \mathit{ret}(r)$
```

```
$\{ \mathit{ret} \lhd \mathbf{cont}(\mathsf{F}; r.\, r \lhd \mathbf{own}\ \mathbf{()}) \}$
  $\mathtt{newlft};$
Local lifetimes: $\alpha \sqsubseteq_1 [\,]$
   $\{\}$ $\mathtt{let}\ t = \mathtt{new}(2)\ \mathtt{in}\ \mathtt{let}\ f = \mathtt{new}(1)\ \mathtt{in}\ \mathtt{let}\ g = \mathtt{new}(1)\ \mathtt{in}$
   $\{ t \lhd \mathbf{own}\ \lightning_2, f \lhd \mathbf{own}\ \lightning_1, g \lhd \mathbf{own}\ \lightning_1 \}$
   $\{f\} f := 42; \{ f \lhd \mathbf{own}\ \mathbf{int} \}$
   $\{f\}$ Borrow at $\alpha$ $\{ f \lhd \&^{\alpha}_{\mathbf{shr}}\ \mathbf{int}, f \lhd^{\dagger\alpha} \mathbf{own}\ \mathbf{int} \}$
   $\{t\}$ $t.0 := f; t.1 := f;$ $\{ t \lhd \mathbf{own}\ \mathit{Two}\langle\alpha\rangle \}$
   $\{t\}$ $\mathtt{let}\ \mathit{t0'} = {}^{*}t.0\ \mathtt{in}$ $\{ \mathit{t0'} \lhd \&^{\alpha}_{\mathbf{shr}}\ \mathbf{int}, t \}$
   $\{\mathit{t0'}\}$ ${}^{*}\mathit{t0'};$ $\{\mathit{t0'}\}$
   $\{g\}$ $g := 23;$ $\{ g \lhd \mathbf{own}\ \mathbf{int} \}$
   $\{g\}$ Borrow at $\alpha$ $\{ g \lhd \&^{\alpha}_{\mathbf{shr}}\ \mathbf{int}, g \lhd^{\dagger\alpha} \mathbf{own}\ \mathbf{int} \}$
   $\{g, t\}$ $t.1 := g;$ $\{g, t\}$
   $\{\}$ $\mathtt{let}\ r = \mathtt{new}(0)\ \mathtt{in}$ $\{ r \lhd \mathbf{own}\ \mathbf{()} \}$
   $\{f, g\}$ $\mathtt{endlft};$ $\{ g \lhd \mathbf{own}\ \mathbf{int}, f \lhd \mathbf{own}\ \mathbf{int} \}$
   $\{t, f, g, \mathit{ret}\}$ $\mathtt{delete}(2, t); \mathtt{delete}(1, f); \mathtt{delete}(1, g); \mathtt{jump}\ \mathit{ret}(r)$
```

3 λ_{Rust} in Iris

Before we get started with the interesting parts, we do some preparatory work on the Iris level to enable reasoning about lambdaRust work. Mostly this is a straight-forward lifting of the operational semantics; the part we need to describe in more details is how we manage the heap.

Physical state. To manage the heap, we use two monoids: Finite partial functions from locations to pairs of lock states and (exclusive) values to talk about ownership of locations (with the instance named γ_{PhVal}), and their contents; and finite partial functions from blocks to (exclusive) tuples for start index and length of the block (instance name γ_{PhFree}). For that to work out, we give a monoid structure to lock states with some unit ε and $\mathtt{\mathbf{reading}}\, n \cdot \mathtt{\mathbf{reading}}\, m = \mathtt{\mathbf{reading}}\, n + m$.

$$
\begin{aligned}
\textsc{PhVal} &:= \mathit{Loc} \overset{\text{fin}}{\rightharpoonup} \textsc{Frac}(\mathit{LockSt} \times \textsc{Ag}(\mathit{Val})) \\
\textsc{PhFree} &:= \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \textsc{Frac}(\textsc{PowFin}(\mathbb{N})) \\
\ell \overset{q}{\mapsto} v &:= \boxed{\circ\,[\ell \leftarrow q(\mathtt{\mathbf{reading}}\, 0, v)] : \textsc{Auth}(\textsc{PhVal})}^{\gamma_{\text{PhVal}}} \\
\ell \mapsto v &:= \ell \overset{1}{\mapsto} v \\
\ell \overset{q}{\mapsto} \overline{v} &:= \mathop{\Huge *}_{i} \ell + i \overset{q}{\mapsto} \overline{v}_i \\
\ell \mapsto \overline{v} &:= \ell \overset{1}{\mapsto} \overline{v} \\
\dagger^{m}_{q}\, \ell &:= \exists i, n.\, \ell = (i, n) \land \boxed{\circ\,[i \leftarrow q([\geq n, < n + m])]}^{\gamma_{\text{PhFree}}} \\
\mathsf{InvPhys} &:= \exists h, V, F.\, \mathsf{Phy}(h) * \boxed{\bullet\, V}^{\gamma_{\text{PhVal}}} * \boxed{\bullet\, F}^{\gamma_{\text{PhFree}}} * \\
&\qquad h = V * (\forall i.\, \mathrm{dom}(h) \cap \{i\} \times \mathbb{N} = F(i))
\end{aligned}
$$

where $\dagger^{m}_{1}\, \ell$ is the permission to deallocate ℓ as block of length m. If the fraction is less than 1, this means that we don't own the entire block yet, and have to obtain more permissions before we can deallocate anything. We assume a global invariant namespace $\mathcal{N}_{\text{Ph}}$, and we assume $\boxed{\mathsf{InvPhys}}^{\mathcal{N}_{\text{Ph}}}$ to be in the global context.

We obtain the usual triples for both atomic and non-atomic memory loads and stores, and (de)allocation, and some useful separations:

$$
\ell \overset{q}{\mapsto} v * \ell \overset{q'}{\longmapsto} v' \Leftrightarrow \ell \overset{q+q'}{\longmapsto} v * v = v' \qquad\qquad \dagger^{m}_{q}\, \ell * \dagger^{m'}_{q'}\, \ell + m \Leftrightarrow \dagger^{m+m'}_{q+q'}\, \ell
$$

$$
\{\mathsf{True}\}\, \mathtt{\mathbf{alloc}}(n)\, \{\ell.\, \exists \overline{v}.\, \ell \mapsto \overline{v} * |\overline{v}| = n * \dagger^{n}_{1}\, \ell\}_{\mathcal{N}_{\text{Ph}}} \qquad\qquad \left\{\ell \mapsto \overline{v} * \dagger^{|\overline{v}|}_{1}\, \ell\right\} \mathtt{\mathbf{free}}(|\overline{v}|, v)\, \{\mathsf{True}\}_{\mathcal{N}_{\text{Ph}}}
$$

$$
\left\{\ell \overset{q}{\mapsto} v\right\} {}^{*\mathtt{\mathbf{sc}}}\ell \left\{v'.\, v' = v * \ell \overset{q}{\mapsto} v\right\}_{\mathcal{N}_{\text{Ph}}} \qquad\qquad \left\{\ell \overset{q}{\mapsto} v\right\} {}^{*}\ell \left\{v'.\, v' = v * \ell \overset{q}{\mapsto} v\right\}_{\mathcal{N}_{\text{Ph}}}
$$

$$
\{\ell \mapsto v\}\, \ell :=_{\mathtt{\mathbf{sc}}} w\, \{v'.\, v' = () * \ell \mapsto w\}_{\mathcal{N}_{\text{Ph}}} \qquad\qquad \{\ell \mapsto v\}\, \ell := w\, \{v'.\, v' = () * \ell \mapsto w\}_{\mathcal{N}_{\text{Ph}}}
$$

$$
\frac{|\overline{v}_1| = |\overline{v}_2| = n}{\left\{\ell_1 \mapsto \overline{v}_1 * \ell_2 \overset{q}{\mapsto} \overline{v}_2\right\} \ell_1 :=_n {}^{*}\ell_2 \left\{\ell_1 \mapsto \overline{v}_2 * \ell_2 \overset{q}{\mapsto} \overline{v}_2\right\}_{\mathcal{N}_{\text{Ph}}}}
$$

4 Lifetime logic

The core principle of lifetimes and borrows has applications beyond type systems. In the following, we develop a logic that includes primitives dealing with lifetimes, using the λ_{Rust} type system as a sample application.

4.1 Proof rules

Intuitively speaking, why ought a type system like the one in §1.4 be sound? What justifies doing a borrow, using that borrow, and obtaining ownership of the original permission again when the borrow ends? The purpose of this section is to develop an intuition for the proof rules of Figure 3, which are used in the soundness proof of the type system. These rules describe the *lifetime logic*.

Splitting ownership in time. The lifetime logic adds a built-in notion of *lifetimes*, and the notion of "owning P borrowed for lifetime κ", written $\&^{\kappa}_{\mathbf{full}}\, P$.

The rule LftL-begin is used to create a new lifetime. At this point, we obtain the token $[\kappa]_1$ which asserts that *we own the lifetime* κ: We know that the lifetime is still running, and we can end it any time by applying the view shift we got. Now, it turns out that we may want multiple parties to be able to witness that κ is ongoing, so we need to be able to split this assertion: $[\kappa]_q$ denotes ownership of the fraction q of κ. Lifetimes can be *intersected* using the $\sqcap$ operator.

We also obtain an update to end the new lifetime again. This makes use of the "update that takes a step", defined as follows:

$$P \mathrel{\Rrightarrow\!\!\ast}^{\mathcal{E}_2}_{\mathcal{E}_1} Q := P \mathrel{-\!\!*} {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} \triangleright {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_1} Q$$

The core operation of the lifetime logic is *borrowing* an assertion P at a given lifetime. Using LftL-borrow, P is split into ownership of P during the lifetime κ (the full borrow), and ownership when κ died (a view shift that lets us "inherit" P from κ). In some sense, we are *splitting ownership along the time axis*: The justification for the separating conjunction is the fact that a lifetime is never both ongoing and has already ended at the same time. Thus, the two parts that we split P into can be treated as disjoint resources: They govern the same part of the (logical and physical) state, but they do so at different points in time.

When a lifetime ends, full borrows at that lifetime are not worth anything any more, a fact that is witnessed by LftL-bor-fake.

Borrowed assertions can still be split and merged, as shown by LftL-bor-sep. To get access to a borrowed assertion, we use LftL-bor-acc-cons. The rule is quite a mouthful, so it is worth looking at the following simpler (derived) version:

$$\langle \&^{\kappa}_{\mathbf{full}}\, P * [\kappa]_q \Longleftrightarrow \triangleright P \rangle_{\mathcal{N}_{\mathrm{lft}}} \tag{1}$$

This lets us *open* full borrows ($\&^{\kappa}_{\mathbf{full}}\, P$) if we can prove that the lifetime is still ongoing, which we do by presenting any fraction of the lifetime token. We obtain $\triangleright P$, but lose access to that token for as long as the full borrow is open, which ensures that we do not end the lifetime while the full borrow is open. Once we re-established $\triangleright P$, we can *close* the full borrow again the get our token back.

The full rule LftL-bor-acc-cons actually lets us close not just with $\triangleright P$, but with any $\triangleright Q$ if we can show that Q entails P through a view shift. Furthermore, that view shift is only actually tun when the lifetime ends, which is witnessed by providing the appropriate token ($[\dagger\kappa]$).

Figure 3: Lifetime logic assertions and proof rules

Notation	Meaning	Timeless	Persistent
$[\kappa]_q$	Fraction q of lifetime token for κ: Witnessing that the lifetime is still ongoing	Yes	No
$[\dagger\kappa]$	Witness confirming that the lifetime κ is dead (*i.e.*, it has ended)	Yes	Yes
$\&^{\kappa}_{\mathbf{full}}\, P$	Ownership of the *full borrow* of P for κ	No	No
$\&^{\kappa}_{i}\, P$	There is an *indexed borrow* named i of P for κ	No	Yes
$[\mathrm{Bor} : i]_q$	Ownership of fraction q of the indexed borrow i	Yes	No

Lifetimes. Lifetimes κ form a cancellable PCM with intersection as the operation ($\sqcap$) and unit ε.

$$\kappa \sqsubseteq \kappa' := \big(\forall q.\, \langle [\kappa]_q \Longleftrightarrow q'.\, [\kappa']_{q'} \rangle_{\mathcal{N}_{\mathrm{lft}}}\big) * \big([\dagger\kappa'] \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} [\dagger\kappa]\big)$$

Lifetime creation and end.

LftL-begin
$\mathsf{True} \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists\kappa.\, [\kappa]_1 * \Box\big([\kappa]_1 \mathrel{=\!\!\!\!\ast}^{\mathcal{N}_{\mathrm{lft}}}_{\emptyset} [\dagger\kappa]\big)$

LftL-tok-fract
$[\kappa]_{q+q'} \Leftrightarrow [\kappa]_q * [\kappa]_{q'}$

LftL-tok-comp
$[\kappa \sqcap \kappa']_q \Leftrightarrow [\kappa]_q * [\kappa']_q$

LftL-tok-unit
$\mathsf{True} \Rightarrow [\varepsilon]_q$

LftL-not-own-end
$[\kappa]_q * [\dagger\kappa] \Rightarrow \mathsf{False}$

LftL-end-comp
$[\dagger\kappa \sqcap \kappa'] \Leftrightarrow [\dagger\kappa] \vee [\dagger\kappa']$

LftL-end-unit
$[\dagger\varepsilon] \Rightarrow \mathsf{False}$

Creating full borrows and using them.

LftL-borrow
$\triangleright P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}}\, P * \big([\dagger\kappa] \mathrel{=\!\!\!\!\ast}_{\mathcal{N}_{\mathrm{lft}}} \triangleright P\big)$

LftL-bor-sep
$\&^{\kappa}_{\mathbf{full}}(P * Q) \Lleftarrow\!\!\!\Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}}\, P * \&^{\kappa}_{\mathbf{full}}\, Q$

LftL-bor-fake
$[\dagger\kappa] \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}}\, P$

LftL-bor-acc-strong
$\&^{\kappa}_{\mathbf{full}}\, P * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists\kappa'.\, \kappa \sqsubseteq \kappa' * \triangleright P * \Big(\forall Q.\, \triangleright\big(\triangleright Q * [\dagger\kappa'] \mathrel{=\!\!\!\!\ast}_{\emptyset} \triangleright P\big) * \triangleright Q \mathrel{=\!\!\!\!\ast}_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa'}_{\mathbf{full}}\, Q * [\kappa]_q\Big)$

LftL-bor-acc-atomic-strong
$\&^{\kappa}_{\mathbf{full}}\, P \;{}^{\mathcal{N}_{\mathrm{lft}}}\!\!\Rrightarrow^{\emptyset} \Big(\exists\kappa'.\, \kappa \sqsubseteq \kappa' * \triangleright P * \Big(\forall Q.\, \triangleright\big(\triangleright Q * [\dagger\kappa'] \mathrel{=\!\!\!\!\ast}_{\emptyset} \triangleright P\big) * \triangleright Q \;{}^{\emptyset}\!\!\mathrel{=\!\!\!\!\ast}^{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa'}_{\mathbf{full}}\, Q\Big)\Big) \vee$
$\Big([\dagger\kappa] * {}^{\emptyset}\!\!\mathrel{|\!\!\!\Rrightarrow}^{\mathcal{N}_{\mathrm{lft}}} \mathsf{True}\Big)$

Indexed borrows.

LftL-bor-idx
$\&^{\kappa}_{\mathbf{full}}\, P \Leftrightarrow \exists i.\, \&^{\kappa}_{i}\, P * [\mathrm{Bor} : i]_1$

LftL-bor-fract
$[\mathrm{Bor} : i]_{q+q'} \Leftrightarrow [\mathrm{Bor} : i]_q * [\mathrm{Bor} : i]_{q'}$

LftL-idx-shorten
$$\frac{\kappa' \sqsubseteq \kappa}{\&^{\kappa}_{i}\, P \Rightarrow \&^{\kappa'}_{i}\, P}$$

LftL-idx-acc
$\&^{\kappa}_{i}\, P \vdash \langle [\mathrm{Bor} : i]_1 * [\kappa]_q \Longleftrightarrow \triangleright P \rangle_{\mathcal{N}_{\mathrm{lft}}}$

LftL-idx-acc-atomic
$\&^{\kappa}_{i}\, P \vdash \langle [\mathrm{Bor} : i]_q \Longleftrightarrow b.\, \text{if } b \text{ then } \triangleright P \text{ else } [\dagger\kappa] \rangle^{\emptyset}_{\mathcal{N}_{\mathrm{lft}}}$

LftL-idx-bor-unnest
$\&^{\kappa}_{i}\, P * \&^{\kappa'}_{\mathbf{full}}([\mathrm{Bor} : i]_1) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa \sqcap \kappa'}_{\mathbf{full}}\, P$

Finally, the rule LftL-bor-acc-atomic-cons provides a way to access a full borrow *without* having a proof that the lifetime is still ongoing. The rule is again fairly involved due to incorporating a rule of consequence, so let us consider the following simpler (derived) version:

$$\langle \&^{\kappa}_{\mathbf{full}}\, P \Lleftarrow\!\!\Rrightarrow b.\, \text{if } b \text{ then } \triangleright P \text{ else } [\dagger\kappa] \rangle^{\emptyset}_{\mathcal{N}_{\text{lft}}} \tag{2}$$

The key differences to is that we do *not* need to provide $[\kappa]_q$. As a consequence, (a) the accessor is mask-changing, so it can only be used atomically, and (b) we may get *either* $\triangleright P$ (the content of the full borrow) *or* a proof that κ has, in fact, already ended.

A closer look at lifetimes. Before we go on talking about the lifetime logic rules, we have to become more concrete about what a *lifetime* κ is. Lifetimes κ form a partial commutative monoid with unit ε. We will also refer to the composition operation ($\sqcap$) as *intersection* of lifetimes. Moreover, the PCM has to be *cancellable*, which means that the composition function is injective.

Furthermore, we define the following inclusion relation on lifetimes:

$$\kappa \sqsubseteq \kappa' := \Box \left(\left(\forall q.\, \langle [\kappa]_q \Lleftarrow\!\!\Rrightarrow q'.\, [\kappa']_{q'} \rangle_{\mathcal{N}_{\text{lft}}} \right) * \left([\dagger\kappa'] \Rrightarrow_{\mathcal{N}_{\text{lft}}} [\dagger\kappa] \right) \right)$$

This says that κ is dynamically shorter than κ' if, given any fraction the token for κ, we can produce some fraction of the token for κ'. Furthermore, tokens showing that κ' has ended must be convertible to tokens showing that κ has ended. It is easy to show that this inclusion interacts as expected with lifetime intersection (LftL-incl-isect).

Indexed borrows. While the proof rules given so far bring us pretty far, it turns out that for some of the advanced reasoning we need to do for Rust, they do not suffice. As we start to build more complicated protocols involving full borrows, the fact that $\&^{\kappa}_{\mathbf{full}}\, P$ is neither timeless nor persistent really becomes a problem.

For this reason, the logic provides a way to *decompose* a full borrow into timeless and persistent pieces (the borrow token and the indexed borrow, respectively), which are tied together by an *index* i (LftL-bor-idx). Indexed borrows can be opened using LftL-idx-acc, but they cannot be strengthened, reborrowed or split. They can also be accessed atomically without a lifetime token LftL-idx-acc-atomic. The latter rule further shows that we actually only need *any fraction* of the borrow token to perform an atomic access, thus providing a way of sharing borrows (by distributing their fractions). Furthermore, indexed borrows can be *shortened* (LftL-idx-shorten) following the dynamic lifetime inclusion $\kappa' \sqsubseteq \kappa$.

Indexed borrows are used to state the rule LftL-idx-bor-unnest, which will be used later to prove two important derived rules: unnesting and reborrowing.

4.2 Derived forms of borrowing

Figure 4 shows some rules that can be derived from the basic rules discussed in the previous subsection. LftL-bor-freeze is a very interesting derived rule, and it also demonstrates the full power of LftL-bor-acc-atomic-cons, so we will briefly discuss it. After applying LftL-bor-acc-atomic-cons, we distinguish two cases. Either we get $\triangleright \exists x \in \tau.\, P$. We then move the $\triangleright$ down into the existential and destruct the latter to obtain an x and $\triangleright P$. We pick $Q := P$ and trivially show that $P \twoheadrightarrow \exists x \in \tau.\, P$. Then we run the closing view shift to finish the proof. In the other case, all we

LftL-incl-isect
$$\kappa \sqcap \kappa' \sqsubseteq \kappa$$

LftL-incl-glb
$$\dfrac{\kappa \sqsubseteq \kappa' \qquad \kappa \sqsubseteq \kappa''}{\kappa \sqsubseteq \kappa' \sqcap \kappa''}$$

LftL-fract-lincl
$$\dfrac{\&^{\kappa}_{\mathbf{frac}}\, q'.\, [\kappa']_{q \cdot q'}}{\kappa \sqsubseteq \kappa'}$$

LftL-bor-shorten
$$\dfrac{\kappa' \sqsubseteq \kappa}{\&^{\kappa}_{\mathbf{full}}\, P \Rightarrow \&^{\kappa'}_{\mathbf{full}}\, P}$$

LftL-reborrow
$$\kappa' \sqsubseteq \kappa \vdash \&^{\kappa}_{\mathbf{full}}\, P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa'}_{\mathbf{full}}\, P * \left([\dagger\kappa'] \mathrel{\equiv\!\!\!\ast}_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}}\, P\right)$$

LftL-bor-unnest
$$\&^{\kappa'}_{\mathbf{full}}(\&^{\kappa}_{\mathbf{full}}\, P) \mathrel{\triangleright\!\!\!\equiv\!\!\!\ast}_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa \sqcap \kappa'}_{\mathbf{full}}\, P$$

LftL-bor-acc-cons
$$\&^{\kappa}_{\mathbf{full}}\, P * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \triangleright P * \forall Q.\, \triangleright\left(\triangleright Q \mathrel{\equiv\!\!\!\ast}_{\emptyset} \triangleright P\right) * \triangleright Q \mathrel{\equiv\!\!\!\ast}_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}}\, Q * [\kappa]_q$$

LftL-bor-acc
$$\langle [\kappa]_q * \&^{\kappa}_{\mathbf{full}}\, P \Longleftrightarrow \triangleright P \rangle_{\mathcal{N}_{\mathrm{lft}}}$$

LftL-bor-acc-atomic
$$\langle \&^{\kappa}_{\mathbf{full}}\, P \Longleftrightarrow b.\, \mathrm{if}\ b\ \mathrm{then} \triangleright P\ \mathrm{else}\ [\dagger\kappa] \rangle^{\emptyset}_{\mathcal{N}_{\mathrm{lft}}}$$

LftL-bor-acc-atomic-cons
$$\&^{\kappa}_{\mathbf{full}}\, P \ {}^{\mathcal{N}_{\mathrm{lft}}}\!\!\Rrightarrow^{\emptyset} \left(\triangleright P * \forall Q.\, \triangleright\left(\triangleright Q \mathrel{\equiv\!\!\!\ast}_{\emptyset} \triangleright P\right) * \triangleright Q \ {}^{\emptyset}\!\!\mathrel{\equiv\!\!\!\ast}^{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}}\, Q\right) \vee [\dagger\kappa] * {}^{\emptyset}\!\!\models\!\!\!\Rrightarrow^{\mathcal{N}_{\mathrm{lft}}} \mathsf{True}$$

LftL-bor-freeze
$$\dfrac{\tau \text{ inhabited}}{(\&^{\kappa}_{\mathbf{full}}\, \exists x : \tau.\, P) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists x : \tau.\, \&^{\kappa}_{\mathbf{full}}\, P}$$

Figure 4: Lifetime logic derived rules

get is $[\dagger\kappa]$. We run the closing view shift, and now we use the fact that τ is inhabited to obtain some x. We pick that x as the witness for our goal, and then use LftL-bor-fake to finish the proof.

Furthermore, we introduce in Figure 5 some derived forms of borrowing – that is, assertions that share are somewhat like $\&^{\kappa}_{\mathbf{full}}\, P$, but not exactly.

Reborrowing. Two The rule LftL-reborrow lets us *reborrow* a $\&^{\kappa}_{\mathbf{full}}\, P$, which means that we can pick some statically shorter lifetime $\kappa' \sqsubseteq \kappa$ and obtain P borrowed at κ'. When κ' ends, we can get our original full borrow back.

The rule LftL-bor-unnest is related. It deals with the case that we have a full borrow of a full borrow ($\&^{\kappa'}_{\mathbf{full}}\, \&^{\kappa}_{\mathbf{full}}\, P$). If we have already opened that full borrow and stripped a way the $\triangleright$ added by opening, then we can use LftL-bor-unnest to "unnest" the full borrow in the sense that we end up with a full borrow at the intersected lifetime ($\&^{\kappa' \sqcap \kappa}_{\mathbf{full}}\, P$).

Both of these rules are *derived* from LftL-idx-bor-unnest.

Persistent borrows. Persistent borrows are a persistent version of borrows. This means that many parties are allowed to get access to its content. In order to avoid reentrant accesses, we can use *two* different mechanisms, giving rise to two flavors of persistent borrows.

Similarly to invariants in Iris, the first possible mechanism is to force only atomic accesses. We then get *atomic persistent borrows*, which are essentially like invariant in Iris with the additional quirk that the invariant is only maintained for the duration of the lifetime of the borrow. They can

Notation	Meaning	Timeless	Persistent
$\&_{\mathbf{at}}^{\kappa/\mathcal{N}} P$	There is a *atomic persistent borrow* of P for κ in namespace $\mathcal{N}$	No	Yes
$\&_{\mathbf{frac}}^{\kappa} \lambda q.\, P$	There is a *fractured borrow* of $\lambda q.\, P$ for κ	No	Yes
$\&_{\mathbf{na}}^{\kappa/p.\mathcal{N}} P$	There is a *non-atomic persistent borrow* of P for κ in non-atomic invariant pool p, namespace $\mathcal{N}$	No	Yes

Atomic persistent borrows

LftL-bor-at

$$\mathcal{N} \mathrel{\#} \mathcal{N}_{\mathrm{lft}} \vdash \&_{\mathbf{full}}^{\kappa} P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&_{\mathbf{at}}^{\kappa/\mathcal{N}} P$$

LftL-bor-lftnamesp

$$\&_{\mathbf{full}}^{\kappa} P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&_{\mathbf{at}}^{\kappa/\mathcal{N}_{\mathrm{lft}}} P$$

LftL-at-acc-atomic

$$\&_{\mathbf{at}}^{\kappa/\mathcal{N}} P \vdash \langle \mathsf{True} \Longleftrightarrow b.\, \text{if } b \text{ then } \triangleright P \text{ else } [\dagger\kappa] \rangle_{\mathcal{N}_{\mathrm{lft}},\mathcal{N}}^{\emptyset}$$

LftL-at-acc

$$\&_{\mathbf{at}}^{\kappa/\mathcal{N}} P \vdash \langle [\kappa]_q \Longleftrightarrow \triangleright P \rangle_{\mathcal{N}_{\mathrm{lft}},\mathcal{N}}^{\mathcal{N}_{\mathrm{lft}}-\mathcal{N}}$$

LftL-at-shorten

$$\frac{\kappa' \sqsubseteq \kappa}{\&_{\mathbf{at}}^{\kappa/\mathcal{N}} P \Rightarrow \&_{\mathbf{at}}^{\kappa'/\mathcal{N}} P}$$

Non-atomic persistent borrows

LftL-bor-na

$$\&_{\mathbf{full}}^{\kappa} P \Rrightarrow_{\mathcal{N}} \square\, \&_{\mathbf{na}}^{\kappa/p.\mathcal{N}} P$$

LftL-na-acc

$$\&_{\mathbf{na}}^{\kappa/p.\mathcal{N}} P \vdash \langle [\kappa]_q * [\mathrm{Na} : p.\mathcal{N}] \Longleftrightarrow \triangleright P \rangle_{\mathcal{N}_{\mathrm{lft}},\mathcal{N}}$$

LftL-na-shorten

$$\frac{\kappa' \sqsubseteq \kappa \qquad \mathcal{N} \sqsubseteq \mathcal{N}'}{\&_{\mathbf{na}}^{\kappa/p.\mathcal{N}} P \Rightarrow \&_{\mathbf{na}}^{\kappa'/p.\mathcal{N}'} P}$$

Fractured borrows

LftL-bor-fracture

$$\&_{\mathbf{full}}^{\kappa} \varPhi(1) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&_{\mathbf{frac}}^{\kappa} \varPhi$$

LftL-fract-acc

$$\frac{\forall q_1, q_2.\, \varPhi(q_1 + q_2) \Leftrightarrow \varPhi(q_1) * \varPhi(q_2)}{\&_{\mathbf{frac}}^{\kappa} \varPhi \vdash \langle [\kappa]_q \Longleftrightarrow q'.\, \triangleright \varPhi(q') \rangle_{\mathcal{N}_{\mathrm{lft}}}}$$

LftL-fract-acc-atomic

$$\&_{\mathbf{frac}}^{\kappa} \varPhi \vdash \langle \mathsf{True} \Longleftrightarrow (b, q).\, \text{if } b \text{ then } \triangleright \varPhi(q) \text{ else } [\dagger\kappa] \rangle_{\mathcal{N}_{\mathrm{lft}}}^{\emptyset}$$

LftL-fract-shorten

$$\frac{\kappa' \sqsubseteq \kappa}{\&_{\mathbf{frac}}^{\kappa} \varPhi \Rightarrow \&_{\mathbf{frac}}^{\kappa'} \varPhi}$$

Figure 5: Lifetime logic derived forms

be defined as follows:

$$\&^{\kappa/\mathcal{N}}_{\mathbf{at}}\, P := \exists i.\ \&^{\kappa}_{i}\, P * (\mathcal{N} \mathrel{\#} \mathcal{N}_{\mathrm{lft}} * \boxed{[\mathrm{Bor} : i]_1}^{\mathcal{N}} \vee \mathcal{N} = \mathcal{N}_{\mathrm{lft}} * \boxed{\exists q.\, [\mathrm{Bor} : i]_q}^{\mathcal{N}_{\mathrm{lft}}})$$

The other possible mechanism is to restrict the persistent borrow to be used in a threaded manner, by using the mechanism of *non-atomic invariants* described in the Iris documentation. The persistent borrows of this other flavor are called *non-atomic persistent borrows.* They can be defined by:

$$\&^{\kappa/p.\mathcal{N}}_{\mathbf{na}}\, P := \exists i.\ \&^{\kappa}_{i}\, P * \mathsf{NaInv}^{p.\mathcal{N}}([\mathrm{Bor} : i]_1)$$

Fractured borrows. A *fractured borrow* is a borrow of a permission $\Phi(q)$ that can be *fractured*, *i.e.*, decomposed according to a fraction:

$$\Phi(q_1 + q_2) \Leftrightarrow \Phi(q_1) * \Phi(q_2)$$

Intuitively, it should be possible to share such a borrow, and still obtain some fraction of Φ via a non-atomic accessor, *i.e.*, $\Phi(q)$ can actually be kept around for non-atomic expressions. This is because even if other threads are concurrently accessing the borrow, they will always leave *some* fraction of Φ in the borrow.

The way this works is that we have an atomic persistent borrow which contains some fraction of Φ, and some fraction of the lifetime token, such that the two fractions add up to 1. When the lifetime is ended, the full token of one of the intersected lifetimes is used up, so there cannot be any piece of the lifetime token within the fractured borrow – so the full $\Phi(1)$ is available. The rule of consequence LftL-bor-acc-strong witnesses this fact by providing $[\dagger\kappa]$ to the view shift that is applied.

Fractured borrows can be defined as follows:

$$\&^{\kappa}_{\mathbf{frac}}\, \Phi := \exists \kappa', \gamma.\, \kappa \sqsubseteq \kappa' * \&^{\kappa'/\mathcal{N}_{\mathrm{lft}}}_{\mathbf{at}}\, \exists q.\, \Phi(q) * \boxed{q}^{\gamma} * (q = 1 \vee [\kappa']_{1-q}))$$

Here, are using the Frac RA.

Fractured borrows are particularly interesting for giving rise to dynamic lifetime inclusion (LftL-fract-lincl).

4.3 Model

We will model lifetimes κ as multisets of *atomic lifetimes* $\Lambda \in \mathbb{N}$, which are just identifiers. This forms a cancellable positive commutative monoid with union for composition and $\emptyset$ as the unit.

We will need the following datatypes and CMRAs:

$$\begin{aligned}
\mathsf{BorSt} &:= \mathsf{in} + \mathsf{open}(q) + \mathsf{rebor}(\kappa)\\
\mathsf{LftSt} &:= \mathsf{LftStAlive} + \mathsf{LftStDead}\\
\textsc{ALft} &:= \textsc{Auth}(\mathbb{N} \xrightarrow{\mathrm{fin}} \textsc{Frac} +_{\lightning} ())\\
\textsc{ILft} &:= \textsc{Auth}(\wp^{\mathrm{fin},+}(\mathbb{N}) \xrightarrow{\mathrm{fin}} \textsc{Ag}(\mathcal{G} \times \mathcal{G} \times \mathcal{G}))\\
\textsc{BorBox} &:= \textsc{Auth}(\mathbb{N} \xrightarrow{\mathrm{fin}} \textsc{Ag}(\mathsf{BorSt}) \times \textsc{Frac})\\
\textsc{Cnt} &:= \textsc{Auth}(\mathbb{N})\\
\textsc{InhBox} &:= \textsc{Auth}(\wp^{\mathrm{fin}}(\mathbb{N}))
\end{aligned}$$

We assume some globally known indices γ_{a} and γ_{i} for managing atomic and intersected lifetimes. The two tokens of the lifetime logic are easily modelled:

$$\begin{aligned}[\kappa]_q &:= \mathop{\Huge *}_{\Lambda \in \kappa} \boxed{\circ\ [\Lambda \leftarrow \mathsf{inl}(q)]}^{\gamma_{\mathrm{a}}} \\ [\dagger\kappa] &:= \exists \Lambda \in \kappa.\, \boxed{\circ\ [\Lambda \leftarrow \mathsf{inr}()]}^{\gamma_{\mathrm{a}}}\end{aligned}$$

We will use the following notation for a view shift that frames assertion P_F:

$$P \Rrightarrow\!\!{*}[P_F]_{\mathcal{E}}\, Q := P * P_F \Rrightarrow\!\!{*}_{\mathcal{E}}\, Q * P_F$$

ILft manages the intersected lifetimes; it just records the ghost names of the state managing those lifetimes. To simplify working with this indirection, we define:

$$\begin{aligned}\mathsf{OwnBor}(\kappa, x) &:= \exists \gamma_{\mathrm{bor}}.\, \boxed{\circ\ [\kappa \leftarrow \gamma_{\mathrm{bor}}, \_, \_]}^{\gamma_{\mathrm{i}}} * \boxed{x}^{\gamma_{\mathrm{bor}}} \\ \mathsf{OwnCnt}(\kappa, x) &:= \exists \gamma_{\mathrm{cnt}}.\, \boxed{\circ\ [\kappa \leftarrow \_, \gamma_{\mathrm{cnt}}, \_]}^{\gamma_{\mathrm{i}}} * \boxed{x}^{\gamma_{\mathrm{cnt}}} \\ \mathsf{OwnInh}(\kappa, x) &:= \exists \gamma_{\mathrm{Inh}}.\, \boxed{\circ\ [\kappa \leftarrow \_, \_, \gamma_{\mathrm{Inh}}]}^{\gamma_{\mathrm{i}}} * \boxed{x}^{\gamma_{\mathrm{Inh}}}\end{aligned}$$

Now we can define the core of the model: the protocols for alive and dead lifetimes. We split the namespace $\mathcal{N}_{\mathrm{lft}}$ into three disjoint sub-namespaces $\mathcal{N}_{\mathrm{mgmt}}$, $\mathcal{N}_{\mathrm{bor}}$, $\mathcal{N}_{\mathrm{inh}}$. Here, $A : \mathbb{N} \xrightarrow{\mathrm{fin}} \mathsf{LftSt}$ is a map indicating the state of the atomic lifetimes, and $I : \wp^{\mathrm{fin},+}(\mathbb{N}) \xrightarrow{\mathrm{fin}} \mathcal{G} \times \mathcal{G} \times \mathcal{G}$ indicates which intersected lifetimes exist.

$$
\begin{aligned}
\mathsf{bor\_to\_box}(s) &:= \begin{cases} \mathsf{full} & \text{if } s = \mathsf{in} \\ \mathsf{empty} & \text{otherwise} \end{cases} \\
\mathsf{LftBorAlive}(\kappa, P_B) &:= \exists B : \mathbb{N} \overset{\mathrm{fin}}{\rightharpoonup} \mathsf{BorSt}.\, \mathsf{OwnBor}(\kappa, \bullet\, [i \leftarrow (B(i), 1) \mid i \in \mathrm{dom}(B)]) \ast \\
&\qquad \mathsf{Box}(\mathcal{N}_{\mathrm{bor}}, P_B, [i \leftarrow \mathsf{bor\_to\_box}(B(i)) \mid i \in \mathrm{dom}(B)]) \ast \\
&\qquad \mathop{\Huge\ast}_{i \in \mathrm{dom}(B)} \begin{Bmatrix} \mathsf{True} & \text{if } B(i) = \mathsf{in} \\ [\kappa]_q & \text{if } B(i) = \mathsf{open}(q) \\ \mathsf{OwnCnt}(\kappa', \circ\, 1) \ast \kappa \subset \kappa' & \text{if } B(i) = \mathsf{rebor}(\kappa') \end{Bmatrix} \\
\mathsf{LftBorDead}(\kappa) &:= \exists B : \wp^{\mathrm{fin}}(\mathbb{N}), P_B : \mathsf{Prop}.\, \mathsf{OwnBor}(\kappa, \bullet\, [i \leftarrow (\mathsf{in}, 1) \mid i \in B]) \ast \\
&\qquad \mathsf{Box}(\mathcal{N}_{\mathrm{bor}}, P_B, [i \leftarrow \mathsf{empty} \mid i \in \mathrm{dom}(B)]) \\
\mathsf{LftInh}(\kappa, P_I, s) &:= \exists E : \wp^{\mathrm{fin}}(\mathbb{N}).\, \mathsf{OwnInh}(\kappa, \bullet\, E) \ast \mathsf{Box}(\mathcal{N}_{\mathrm{inh}}, P_I, [i \leftarrow s \mid i \in E]) \\
\mathsf{LftVs}(\kappa, P_B, P_I) &:= \exists n : \mathbb{N}.\, \mathsf{OwnCnt}(\kappa, \bullet\, n) \ast \forall I : \wp^{\mathrm{fin},+}(\mathbb{N}) \overset{\mathrm{fin}}{\rightharpoonup} \mathcal{G} \times \mathcal{G} \times \mathcal{G}. \\
&\triangleright P_B \ast [\dagger\kappa] \Rrightarrow\!\!\ast \left[\mathsf{LftBorDead}(\kappa) \ast \boxed{\bullet\, I}^{\gamma_{\mathrm{i}}} \ast \mathop{\Huge\ast}_{\substack{\kappa' \in \mathrm{dom}(I) \\ \kappa' \subset \kappa}} \mathsf{LftAlive}(\kappa') \right]_{\mathcal{N}_{\mathrm{bor}}} \quad \triangleright P_I \ast \mathsf{OwnCnt}(\kappa \circ n) \\
\mathsf{LftAlive}(\kappa) &:= \exists P_B, P_I.\, \mathsf{LftBorAlive}(\kappa, P_B) \ast \mathsf{LftVs}(\kappa, P_B, P_I) \ast \mathsf{LftInh}(\kappa, P_I, \mathsf{empty}) \\
\mathsf{LftDead}(\kappa) &:= \exists P_I.\, \mathsf{LftBorDead}(\kappa) \ast \mathsf{OwnCnt}(\kappa, \bullet\, 0) \ast \mathsf{LftInh}(\kappa, P_I, \mathsf{full}) \\
\mathsf{LftAliveIn}(A, \kappa) &:= \forall \Lambda \in \kappa.\, A(\Lambda) = \mathsf{LftStAlive} \\
\mathsf{LftDeadIn}(A, \kappa) &:= \exists \Lambda \in \kappa.\, A(\Lambda) = \mathsf{LftStDead} \\
\mathsf{LftInv}(A, \kappa) &:= \mathsf{LftAlive}(\kappa) \ast \mathsf{LftAliveIn}(A, \kappa) \vee \mathsf{LftDead}(\kappa) \ast \mathsf{LftDeadIn}(A, \kappa)
\end{aligned}
$$

Notice that LftAlive and LftVs are defined mutually recursively, which is well-defined because the size of the lifetime κ gets strictly smaller.

The rough idea behind this setup is as follows: For every lifetime κ, we have two boxes: one tracking the borrows, and one tracking the inheritances. The latter are used to obtain resources from dead lifetimes, which is necessary to show the view shift obtained via LftL-borrow and LftL-reborrow. The borrow box contains assertion P_B, and we have an authoritative map B tracking which slices of the box are full and which are not. There are two ways a slice can be empty: either the borrow currently open, and some fraction of the lifetime token was put in here as a deposit. Alternatively, the borrow can be reborrowed to a strictly shorter lifetime (*i.e.*, a lifetime represented by a strictly larger multiset). Ownership of the fragments of B permit changing the state of the slice or removing it (*e.g.*, for splitting, where one slice is removed and two new ones are added). On the inheritance side, the box overall contains assertion P_I. LftInh ensures that the slices of the box are all in the same state, but there is still an authoritative set that manages ownership of slices for the purpose of removing them from the box. Finally, P_B and P_I are connected through LftVs, which roughly speaking says that one can view shift from P_B to P_I. This view shift is executed when a lifetime ends, after which P_I can be used to fill all the inheritance boxes. All that extra machinery in LftVs is needed to support reborrowing.

Based on this, we define LftLCtx, which we always assume to be in the context for the lifetime logic rules.

$$\begin{aligned}
\mathsf{LftLInv} &:= \exists A, I.\, \boxed{\bullet\, A}^{\gamma_{\text{a}}} * \boxed{\bullet\, I}^{\gamma_{\text{i}}} * \mathop{\Huge\ast}_{\kappa \in \text{dom}(I)} \mathsf{LftInv}(A, \kappa) \\
\mathsf{LftLCtx} &:= \boxed{\mathsf{LftInv}}^{\mathcal{N}_{\text{mgmt}}}
\end{aligned}$$

Now we can model the remaining assertions of the logic, and an assertion RawBor ("raw borrows") that will be useful later in the proofs. The indices i in the lifetime logic rules are actually pairs of a lifetime κ' and a box index i.

$$\begin{aligned}
\kappa \sqsubseteq \kappa' &:= \big(\forall q.\, \langle [\kappa]_q \Longleftrightarrow q'.\, [\kappa']_{q'} \rangle_{\mathcal{N}_{\text{lft}}}\big) * \big([\dagger\kappa'] \Rrightarrow_{\mathcal{N}_{\text{lft}}} [\dagger\kappa]\big) \\
[\text{Bor} : \kappa', i]_q &:= \mathsf{OwnBor}(\kappa, \circ\, i \leftarrow (\mathsf{in}, q)) \\
\&^{\kappa}_{\kappa', i}\, P &:= \kappa \sqsubseteq \kappa' * \mathsf{BoxSlice}(\mathcal{N}_{\text{bor}}, P, i) \\
\mathsf{RawBor}(\kappa, P) &:= \exists i.\, \mathsf{BoxSlice}(\mathcal{N}_{\text{bor}}, P, i) * [\text{Bor} : \kappa, i]_1 \\
\&^{\kappa}_{\mathbf{full}}\, P &:= \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \mathsf{RawBor}(\kappa', P)
\end{aligned}$$

Some proof rules are trivially justified: LftL-tok-fract, LftL-tok-comp, LftL-tok-unit, LftL-not-own-end, LftL-end-comp, LftL-end-unit, LftL-bor-idx, LftL-bor-fract, LftL-idx-shorten are all simple implications. We briefly discuss the most important steps of most the remaining rules.[1] When we have a full borrow $\&^{\kappa}_{\mathbf{full}}\, P$, we will call the actual lifetime of the borrow (the one in the existential quantifier) κ_0. In particular, $\kappa \sqsubseteq \kappa_0$.

Proof sketch of LftL-borrow. First we have to check whether κ is already allocated in I; if not, that's easy to do:

$$\mathsf{True} \Rrightarrow_{\mathcal{N}_{\text{lft}}} \boxed{\circ\, [\kappa \leftarrow \_, \_, \_]}^{\gamma_{\text{i}}} \tag{3}$$

The proof of this also extends A with new atomic lifetimes as necessary. In the following, we assume to have this in the context.

Next we take a look at $\mathsf{LftInv}(A, \kappa)$. If the lifetime is dead, we use the following to create a "fake" full borrow:

$$\triangleright^b\, \mathsf{LftBorDead}(\kappa) \Rrightarrow_{\mathcal{N}_{\text{bor}}} \triangleright^b\, \mathsf{LftBorDead}(\kappa) * \mathsf{RawBor}(\kappa, P) \tag{4}$$

$$\triangleright \mathsf{LftInh}(\kappa, P_I, \mathsf{full}) * \triangleright P \Rrightarrow_{\mathcal{N}_{\text{inh}}} \triangleright \mathsf{LftInh}(\kappa, P_I * P, \mathsf{full}) * \big([\dagger\kappa] \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\text{lft}}} \triangleright P\big) \tag{5}$$

(4) just extends the B in $\mathsf{LftBorDead}$ with a fresh element and creates an empty slice in the box. (5) allocates a fresh element in E.

If the lifetime is alive, we use:

$$\triangleright \mathsf{LftInh}(\kappa, P_I, \mathsf{empty}) \Rrightarrow_{\mathcal{N}_{\text{inh}}} \triangleright \mathsf{LftInh}(\kappa, P_I * P, \mathsf{empty}) * \exists j.\, \mathsf{OwnInh}(\kappa, \circ\, \{j\}) * \mathsf{BoxSlice}(\mathcal{N}_{\text{inh}}, P, j) \tag{6}$$

$$\triangleright \mathsf{LftBorAlive}(\kappa, P_B) * \triangleright P \Rrightarrow_{\mathcal{N}_{\text{bor}}} \triangleright \mathsf{LftBorAlive}(\kappa, P_B * P) * \&^{\kappa}_{\mathbf{full}}\, P \tag{7}$$

$$\triangleright \mathsf{LftVs}(\kappa, P_B, P_I) \Rightarrow \triangleright \mathsf{LftVs}(\kappa, P_B * P, P_I * P) \tag{8}$$

[1] The proof of reborrowing is not covered here; it can (like the others) be found in the Coq development.

These are all easy consequences of boxing lemmas and allocating in the authoritative B and E. All that is left to do is show the view shift $[\dagger\kappa] \Rrightarrow\!\!\ast \triangleright P$. From $[\dagger\kappa]$ we learn that κ has ended, so we can get access to $\mathsf{LftDead}(\kappa)$. We can now apply the following lemma to the resources we got from (6):

$$\triangleright \mathsf{LftInh}(\kappa, P_I, \mathsf{full}) * \mathsf{OwnInh}(\kappa, \circ\, \{j\}) * \mathsf{BoxSlice}(\mathcal{N}_{\text{inh}}, P, j) \Rrightarrow_{\mathcal{N}_{\text{inh}}} \triangleright P * \exists P'_I.\ \triangleright \mathsf{LftInh}(\kappa, P'_I, \mathsf{full}) \tag{9}$$

This is shown easily by removing the slice from the box.

Proof sketch of LftL-bor-fake. This is a trivial consequence of (4).

Proof sketch of LftL-bor-acc-cons, LftL-idx-acc. These two work fairly similarly. First we run the accessor in $\kappa \sqsubseteq \kappa_0$ to obtain a token for the actual lifetime κ_0 of the borrow. We have a witness of κ_0 being in I, so we can get $\mathsf{LftInv}(A, \kappa_0)$. Since we have a token $[\kappa_0]_{q'}$, we can get $\mathsf{LftAlive}(\kappa_0)$. All we care about is $\mathsf{LftBorAlive}$, using the following lemma:

$$\begin{aligned}\mathsf{BoxSlice}(\mathcal{N}_{\text{bor}}, P, i) \vdash\ & \triangleright \mathsf{LftBorAlive}(\kappa, P_B) * \mathsf{OwnBor}(\kappa, \circ\, i \leftarrow \mathsf{in}, \mathsf{inl}(\mathsf{ex}())) * [\kappa]_q \Lleftarrow\!\!\Rrightarrow_{\mathcal{N}_{\text{bor}}} \\ & \triangleright \mathsf{LftBorAlive}(\kappa, P_B) * \mathsf{OwnBor}(\kappa, \circ\, i \leftarrow \mathsf{open}(q), \mathsf{inl}(\mathsf{ex}())) * \triangleright P\end{aligned} \tag{10}$$

From $B(i) = \mathsf{in}$ we know that the box slice is full. We empty the slice, obtaining $\triangleright P$. Next, we change $B(i)$ to $\mathsf{open}(q)$. This allows us to re-establish $\mathsf{LftBorAlive}$. Similar, for the right-to-left direction, we fill the empty slice and change $B(i)$ back to in.

This already pretty much completes the proof of LftL-idx-acc. For the closing view shift, we follow the same path, using the right-to-left direction of (10).

To finish LftL-bor-acc-cons, we have to handle the rule of consequence embedded in the closing view shift of the accessor. We start out like above, until we get $\mathsf{LftBorAlive}$. We use remove the empty slice from the borrow box to learn that P_B later decomposes into $P'_B * P$. Since $\mathsf{LftBorAlive}$ and LftVs are contractive in P_B, we can rewrite with this decomposition. After adding a new empty slice to the box, we can obtain $\triangleright \mathsf{LftBorAlive}(\kappa_0, P'_B * Q)$ (this also requires removing the old slice from the authoritative map and adding a new one, but we own the fragment, so that's possible). Then we can finish the proof by applying (10) and

$$\triangleright \mathsf{LftVs}(\kappa, P'_B * P) * \triangleright \left(\triangleright Q * [\dagger\kappa] \Rrightarrow\!\!\ast_{-\mathcal{N}_{\text{lft}}} \triangleright P\right) \Rightarrow \triangleright \mathsf{LftVs}(\kappa, P'_B * Q) \tag{11}$$

This last lemma follows by composing the view shift in LftVs with the one we get (from Q to P).

Proof sketch of LftL-bor-acc-atomic-cons, LftL-idx-acc-atomic. We start by inspecting whether κ_0 is alive. If it is not, we obtain $[\dagger\kappa_0]$, close the invariant again, and use $\kappa \sqsubseteq \kappa_0$ to obtain $[\dagger\kappa]$. The closing view shift is trivial.

If κ' is alive, we take apart $\triangleright \mathsf{LftBorAlive}(\kappa, P_B)$. From the borrow token we got, we have $B(i) = \mathsf{in}$, and thus the slice with P is full and we can empty it.

We are done with LftL-idx-acc-atomic now: for the closing view shift, we get $\triangleright P$ again, so we fill the slice and are done.

For LftL-bor-acc-atomic-cons, we instead continue like we did above with LftL-bor-acc-cons: we remove the empty slice and create a new one with Q in it, and then we update LftVs using (11).

Proof sketch of LftL-bor-sep. First, we look at the left-to-right direction (splitting), which is simpler. We start by looking at $\mathsf{LftInv}(\kappa')$. If it is dead, then we use (4) to just “fake” $\&^{\kappa'}_{\textbf{full}} P$ and $\&^{\kappa'}_{\textbf{full}} Q$, which can both be changed to borrows at κ using LftL-idx-shorten.

So we can assume we have $\triangleright \mathsf{LftAlive}(\kappa')$ and, in particular, $\triangleright \mathsf{LftBorAlive}$. We split the slice containing $P * Q$ into two slices containing P and Q, respectively. We also have to fix up the authoritative map B, which is easy because we own the fragment corresponding to the slice we removed. This gives us fragments for the new new slices, so we can finish up the proof by putting these fragments into the proofs of $\&^{\kappa'}_{\textbf{full}} P$ and $\&^{\kappa'}_{\textbf{full}} Q$.

Next, we look at the right-to-left direction (merging). The trouble here is that we obtain *two* borrows, at two potentially different lifetimes κ_0 and κ_1. We do have $\kappa \sqsubseteq \kappa_0$ and $\kappa \sqsubseteq \kappa_1$. We use reborrowing to obtain both borrows at lifetime $\kappa' := \kappa \sqcap \kappa_0 \sqcap \kappa_1$. Notice that we have $\kappa \sqsubseteq \kappa'$ Now we can start to do the actual merging. We check whether κ' is dead; if yes, so is κ and we can “fake” the result. So we obtain $\triangleright \mathsf{LftAlive}(\kappa')$ and, in particular, $\triangleright \mathsf{LftBorAlive}$. We merge the two slices containing P and Q, respectively, and turn them into a slice containing $P * Q$. We have all the fragments we need to fix up the authoritative B, so we can close everything up again and obtain $\&^{\kappa'}_{\textbf{full}} P * \&^{\kappa'}_{\textbf{full}} P$. By LftL-bor-shorten, we are done.

Proof sketch of LftL-begin. The first step of this proof is easy, it just involves allocating a new atomic lifetime Λ in ALft and returning a singleton $\kappa := \{\Lambda\}$.

This leaves us with proving the closing view shift. Before we come to the core proof, we show some helper lemmas. We start with a lemma to end a single intersected lifetime κ:

$$\begin{aligned}&(\forall \kappa'.\, \kappa' \in \mathrm{dom}(I) \land \kappa' \subset \kappa \Rightarrow \kappa' \in K) \land (\forall \kappa'.\, \kappa' \in \mathrm{dom}(I) \land \kappa \subset \kappa' \Rightarrow \kappa' \in K') \vdash \\ &\mathsf{LftAlive}(\kappa) * [\dagger\kappa] \Rrightarrow\!\!* \left[\boxed{\bullet\, I}^{\gamma_{\mathrm{i}}} * \mathop{\Huge *}_{\kappa' \in K} \mathsf{LftAlive}(\kappa') * \mathop{\Huge *}_{\kappa' \in K'} \mathsf{LftDead}(\kappa') \right]_{-\mathcal{N}_{\mathrm{mgmt}}} \qquad \mathsf{LftDead}(\kappa)\end{aligned} \tag{12}$$

The proof proceeds by taking a closer look at $\mathsf{LftBorAlive}$ (in $\mathsf{LftAlive}(\kappa)$). The goal is to show that all $B(i)$ are in. We can rule out open because we have $[\dagger\kappa]$. For $\mathsf{rebor}(\kappa')$, we obtain $\mathsf{OwnCnt}(\kappa', \circ\, 1)$ for κ' strictly larger than κ. However, this implies $\kappa' \in K'$ and thus we have $\mathsf{LftDead}(\kappa')$, which says that the authoritative count is 0 (and the ghost names match) – a contradiction. This we know $B(i) = \mathsf{in}$. After emptying the borrow box, we obtain $\triangleright P_B$ and $\mathsf{LftBorDead}(\kappa)$. Next, we apply LftVs. We have all its preconditions: $\triangleright P_B$, $[\dagger\kappa]$, the authoritative I and $\mathsf{LftAlive}$ for all strictly shorter lifetimes. We obtain $\triangleright P_I$ and $\mathsf{OwnCnt}(\kappa, \circ\, n)$, which we use with the authoritative counter to set that to 0. We fill the inheritance box, obtaining $\mathsf{LftInh}(\kappa, P_I, \mathsf{full})$ which completes the proof.

Next, we show how to end a whole set K of lifetimes at once. To this end, the set K must be closed under smaller lifetimes, *i.e.*, larger sets. Furthermore, we need a proof that all all the other lifetimes in I that have some sublifetime in K that's alive according to A, are still alive (these lifetimes are collected in K').

$$\begin{aligned}&(\forall \kappa \in K.\, \forall \kappa' \in \mathrm{dom}(I).\, \kappa' \supseteq \kappa \Rightarrow \kappa' \in K) \land \\ &(\forall \kappa \in K.\, \forall \kappa' \in \mathrm{dom}(I).\, \mathsf{LftAliveIn}(A, \kappa) \land \kappa' \notin K \land \kappa' \subset \kappa \Rightarrow \kappa' \in K') \vdash \\ &\left(\mathop{\Huge *}_{\kappa \in K} \mathsf{LftInv}(A, \kappa) * [\dagger\kappa] \right) \Rrightarrow\!\!* \left[\boxed{\bullet\, I}^{\gamma_{\mathrm{i}}} * \mathop{\Huge *}_{\kappa' \in K'} \mathsf{LftAlive}(\kappa') \right]_{-\mathcal{N}_{\mathrm{mgmt}}} \mathop{\Huge *}_{\kappa \in K} \mathsf{LftDead}(\kappa)\end{aligned} \tag{13}$$

To show (13), we perform induction over the metric $|K|$, *i.e.*, over the size of the kill-set K. We start by checking whether K contains any lifetimes κ such that $\mathsf{LftAliveIn}(A, \kappa)$. If no, we have nothing to do. Otherwise, we select a *minimal* element $\kappa \in \{\kappa \in K \mid \mathsf{LftAliveIn}(A, \kappa)\}$ according to the relation $\subset$. This relation is acyclic, so such an element has to exist. We let $K'' := K \setminus \{\kappa\}$ and $K''' := K' \cup \{\kappa\}$. Clearly, K'' is smaller than K, so we can invoke the induction hypothesis to kill K'' while framing K'. To this end, we have to show that K'' is up-closed. This is the case because κ is not only minimal in $\{\kappa \in K \mid \mathsf{LftAliveIn}(A, \kappa)\}$, it is also minimal in K. Furthermore we have to show that K''' contains all lifetimes below something alive in K''. This is the case, because such a lifetime κ' will either also be in the down-closure of K (and hence it is in K'), or it will be κ itself, which we added to K'''. After invoking the induction hypothesis, we have that all lifetimes in K'' are dead. To complete our goal, all that's left to do is end κ. To this end, we invoke (12). By our frame and by the fact that κ is a minimal alive lifetime in K, we know that all lifetimes strictly shorter than κ are in K' and hence alive. Furthermore, we know that all lifetimes strictly larger than κ are in K'' and hence dead. This completes the proof.

Now, we can come back to proving the closing view shift of LftL-begin. We start out assuming $[\{\Lambda\}]_1$. We open $\mathsf{LftLCtx}$ and take a step to obtain $\mathsf{LftLInv}$. From the token we own, we know $A(\Lambda) = \mathsf{LftStAlive}$. We update our token to obtain $[\dagger\,\{\Lambda\}]$. Now we want to apply (13) with $K := \{\kappa \in I \mid \Lambda \in \kappa\}$ and $K' := \{\kappa \in \mathrm{dom}(I) \mid \kappa \notin K \wedge \exists \kappa' \in K.\, \mathsf{LftAliveIn}(A, \kappa') \wedge \kappa \subset \kappa'\}$. From $[\dagger\,\{\Lambda\}]$ we can get the corresponding token for all $\kappa \in K$. We thus satisfy all requirements of (13), which finishes the proof.

5 λ_{Rust} model

Well-formed terms of the type system are interpreted as Iris terms. Valuable expressions are interpreted as the value they evaluate to. Variables in the type system are interpreted as Iris variables of appropriate sort.

5.1 Types

Types are complex beasts. After the preparations we made in §4, it should not come as a surprise that borrowing of λ_{Rust} types will be explained using the lifetime logic. The notation has already been suggestively chosen to match the one used in the syntactic type system.

The domain of semantic types is defined in Figure 6, and the interpretation of all primitive types is defined in Figure 7. In the following, we will develop this definition step-by-step, together with the semantic interpretation of the most important (and most complex) types: Owned pointers, as well as mutable and shared references. We will also talk about products (restricted to the representative case of pairs), which is probably the least surprising type.

The core of a semantic type is its notion of *ownership* $[\![\tau]\!].\mathrm{own} : \mathit{TId} \times \mathit{list}(\mathit{Val}) \to \mathit{iProp}$. This is a *thread-indexed predicate over a list of values*. Why *lists* of values? A type describes a *continuous region of memory*: Compound types like products and sums take up more than one memory location, because they need more than one value to be represented. For example, the interpretation of $\tau_1 \times \tau_2$ will demand that the given list is the *concatenation* of two lists accepted by τ_1 and τ_2, respectively, while the uninitialized type just accepts any list of appropriate length:

$$\begin{aligned}
[\![\tau_1 \times \tau_2]\!].\mathrm{own}(t, \overline{v}) &:= \exists \overline{v_1}, \overline{v_2}.\, \overline{v} = \overline{v_1} +\!\!\!+\, \overline{v_2} * [\![\tau_1]\!].\mathrm{own}(t, \overline{v_1}) * [\![\tau_2]\!].\mathrm{own}(t, \overline{v_2}) \\
[\![\lightning_n]\!].\mathrm{own}(\_, \overline{v}) &:= |\overline{v}| = n
\end{aligned}$$

The thread-relative nature of these predicates is needed to model types which are not `Send` or `Sync`, types which crucially rely on being accessed from only a single thread.

It turns out that we need another bit of data to properly describe a type: We need to know its *size*. This is given as $[\![\tau]\!].\mathrm{size} : \mathbb{N}$, such that Ty-size holds. Note that we also need to have

Semantic Type A *semantic type* is a tuple $(\mathrm{size} \in \mathbb{N}, \mathrm{own} \in \mathit{TId} \times \mathit{list}(\mathit{Val}) \to \mathit{iProp}, \mathrm{shr} \in \mathit{Lft} \times \mathit{TId} \times \mathit{Loc} \to \mathit{iProp})$ such that

$$\forall t, \overline{v}.\, \mathrm{own}(t, \overline{v}) \Rightarrow |\overline{v}| = \mathrm{size} \qquad (\textsc{Ty-size})$$

$$\forall \kappa, t, \ell.\, \mathrm{shr}(\kappa, t, \ell) \Rightarrow \Box\, \mathrm{shr}(\kappa, t, \ell) \qquad (\textsc{Ty-shr-persistent})$$

$$\forall \kappa, t, \ell.\ \&^{\kappa}_{\mathbf{full}}\, (\ell \mapsto \mathrm{own}(t)) * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \mathrm{shr}(\kappa, t, \ell) * [\kappa]_q \qquad (\textsc{Ty-share})$$

$$\forall \kappa, \kappa', t, \ell.\, \kappa' \sqsubseteq \kappa \Rightarrow \mathrm{shr}(\kappa, t, \ell) \Rightarrow \mathrm{shr}(\kappa', t, \ell) \qquad (\textsc{Ty-shr-mono})$$

Semantic type inclusion

$$\begin{aligned}
\tau_1 \sqsubseteq^{\mathsf{ty}} \tau_2 := \tau_1.\mathrm{size} = \tau_2.\mathrm{size} \wedge (\Box\, \forall t, \overline{v}.\, \tau_1.\mathrm{own}(t, \overline{v}) \Rightarrow \tau_2.\mathrm{own}(t, \overline{v})) \wedge \\
(\Box\, \forall \kappa, t, \ell.\, \tau_1.\mathrm{shr}(\kappa, t, \ell) \Rightarrow \tau_2.\mathrm{shr}(\kappa, t, \ell))
\end{aligned}$$

Figure 6: The semantic domain of types.

$\mathsf{size}(\tau) = [\![\tau]\!].\mathrm{size}$.

5.1.1 Owned pointers and mutable references

With the foundations introduced above, we can now define

$$\begin{aligned}
[\![\mathbf{\mathsf{own}}_n\, \tau]\!].\mathrm{own}(t, \overline{v}) &:= \exists \ell.\, \overline{v} = [\ell] * \triangleright \ell \mapsto [\![\tau]\!].\mathrm{own}(t) * \left([\![\tau]\!].\mathrm{size} = 0 \vee [\![n]\!] > 0 * \triangleright \dagger^{[\![\tau]\!].\mathrm{size}}_{[\![\tau]\!].\mathrm{size}/[\![n]\!]}\, \ell\right) \\
[\![\&^{\kappa}_{\mathbf{\mathsf{mut}}}\, \tau]\!].\mathrm{own}(t, \overline{v}) &:= \exists \ell.\, \overline{v} = [\ell] * \&^{\kappa}_{\mathbf{\mathsf{full}}}\, \ell \mapsto [\![\tau]\!].\mathrm{own}(t)
\end{aligned}$$

Here and in the remainder of this document, $\ell \mapsto \Phi$ is sugar for $\exists \overline{v}.\, \ell \mapsto \overline{v} * \Phi(\overline{v})$. Notice the close relationship between the two types: The mutable reference has a borrow of the content where the owned pointer, well, owns it. In particular, both definitions are *contractive*. The only remaining difference is that owned pointers have the permission to deallocate what they point to, which mutable references do not have.

With this setup, it is clear how borrowing can start (C-borrow). It is also clear that owning a mutable reference *at an ongoing lifetime* justifies reading and writing through that pointer (Tread-bor and Twrite-bor).

Re-borrowing an already borrowed pointer (C-reborrow) is justified by LftL-reborrow.

The most interesting rules are S-deref-bor-own and S-deref-bor-bor. In both cases, after opening the borrow, we *freeze* the current value of the pointer: Since the original borrow is lost, we know it cannot be changed anymore. For S-deref-bor-bor, we also have to justify getting out the inner borrow: We start with a nested borrow, and after opening the outer one and taking a step, we use LftL-bor-unnest to both close the outer borrow and obtain the inner one at the outer lifetime.

5.1.2 Shared references

The story is unfortunately more complicated for shared references. This is because sharing is a rather non-uniform idea in Rust: For many types, sharing them (*i.e.*, having a shared reference to an element of such a type) implies that the contents of the variable are *frozen*, *i.e.*, all mutation is prohibited. This makes it trivially safe for everybody to perform read-only actions on this variable, without and risk of data races or invalid pointers. The obvious model for this in separation logic is to provide every holder of a shared reference with some fraction of the pointer.

Other types, however, provide *interior mutability*, which means that it *is* possible to perform mutation through a shared reference. This should be modeled by having *full* ownership of the pointer available, guarded through some protocol so that every holder of a shared reference can obtain the full permission if they follow the protocol.

The way we express this formally is that *every type decides itself what it means to be shared.* The interpretation of $\&^{\kappa}_{\mathbf{\mathsf{shr}}}\, \tau$ is not given uniformly; instead, it is defined by the type itself:

$$[\![\&^{\kappa}_{\mathbf{\mathsf{shr}}}\, \tau]\!].\mathrm{own}(t, \overline{v}) := \exists \ell.\, \overline{v} = [\ell] * \Box\, [\![\tau]\!].\mathrm{shr}(\kappa, t, \ell)$$

The sharing predicate has to satisfy two core properties: By Ty-shr-persistent, sharing is persistent. By Ty-share, sharing can be started from a mutable reference (compare the premise of this rule to the interpretation of mutable references above). Furthermore, sharing predicates have to satisfy some closure properties (Ty-shr-mono).

$$\begin{aligned}
\llbracket \mathbf{!} \rrbracket_\gamma &:= \mathrm{Type}\,\{\mathrm{size} := 0; \mathrm{own} := \lambda\_,\_.\, \mathsf{False}; \mathrm{shr} := \lambda\_,\_,\_.\, \mathsf{False}\} && (\textsc{Ty-def-emp}) \\
\llbracket \mathbf{()} \rrbracket_\gamma &:= \mathrm{Type}\,\{\mathrm{size} := 0; \mathrm{own} := \lambda\_,\overline{v}.\, \overline{v} = []; \mathrm{shr} := \lambda\_,\_,\_.\, \mathsf{True}\} && (\textsc{Ty-def-unit}) \\
\llbracket \textsf{\textbf{bool}} \rrbracket_\gamma &:= \mathrm{SimpleType}(\lambda\_, v.\, v = \texttt{\textbf{true}} \vee v = \texttt{\textbf{false}}) && (\textsc{Ty-def-bool}) \\
\llbracket \textsf{\textbf{int}} \rrbracket_\gamma &:= \mathrm{SimpleType}(\lambda\_, \overline{v}.\, \exists z.\, v = z) && (\textsc{Ty-def-nat})
\end{aligned}$$

$$\begin{aligned}
\llbracket \textsf{\textbf{own}}_n\, \tau \rrbracket_\gamma := \mathrm{Type}\,\{&\mathrm{size} := 1; \\
&\mathrm{own} := \lambda t, \overline{v}.\, \exists \ell.\, \overline{v} = [\ell] * \triangleright \ell \mapsto \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t) * \\
&\qquad \Big(\llbracket \tau \rrbracket_\gamma.\mathrm{size} = 0 \vee \llbracket n \rrbracket_\gamma > 0 * \triangleright \dagger^{\llbracket \tau \rrbracket_\gamma.\mathrm{size}}_{\llbracket \tau \rrbracket_\gamma.\mathrm{size}/\llbracket n \rrbracket_\gamma} \ell \Big); \\
&\mathrm{shr} := \lambda \kappa, t, \ell.\, \exists \ell'.\, \&^{\kappa}_{\mathbf{frac}}(\lambda q'.\, \ell \xmapsto{q'} \ell') * \\
&\qquad \Box\big(\forall q'.\, [\kappa]_{q'} \Rrightarrow\!\!\ast^{\mathcal{N}_{\mathrm{lft}}}_{\mathcal{N}_{\mathrm{shr}},\mathcal{N}_{\mathrm{lft}}} \llbracket \tau \rrbracket_\gamma.\mathrm{shr}(\kappa, t, \ell') * [\kappa]_{q'}\big)\} \\
& && (\textsc{Ty-def-own})
\end{aligned}$$

$$\begin{aligned}
\llbracket \&^{\kappa}_{\textsf{\textbf{mut}}}\, \tau \rrbracket_\gamma := \mathrm{Type}\,\{&\mathrm{size} := 1; \\
&\mathrm{own} := \lambda t, \overline{v}.\, \exists \ell.\, \overline{v} = [\ell] * \&^{\llbracket \kappa \rrbracket}_{\mathbf{full}}\, \ell \mapsto \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t); \\
&\mathrm{shr} := \lambda \kappa', t, \ell.\, \exists \ell'.\, \&^{\kappa'}_{\mathbf{frac}}(\lambda q.\, \ell \xmapsto{q} \ell') * \\
&\qquad \Box\big(\forall q.\, [\llbracket \kappa \rrbracket \sqcap \kappa']_q \Rrightarrow\!\!\ast^{\mathcal{N}_{\mathrm{lft}}}_{\mathcal{E},\mathcal{N}_{\mathrm{lft}}} \llbracket \tau \rrbracket_\gamma.\mathrm{shr}(\llbracket \kappa \rrbracket \sqcap \kappa', t, \ell') * [\llbracket \kappa \rrbracket \sqcap \kappa']_q\big)\} \\
& && (\textsc{Ty-def-mut})
\end{aligned}$$

$$\llbracket \&^{\kappa}_{\textsf{\textbf{shr}}}\, \tau \rrbracket_\gamma := \mathrm{SimpleType}(\lambda t, v.\, \exists \ell.\, v = \ell * \llbracket \tau \rrbracket_\gamma.\mathrm{shr}(\llbracket \kappa \rrbracket, t, \ell)) \qquad (\textsc{Ty-def-shr})$$

$$\begin{aligned}
\llbracket \Pi \overline{\tau} \rrbracket_\gamma := \mathrm{Type}\,\{&\mathrm{size} := \sum_i \llbracket \overline{\tau}_i \rrbracket_\gamma.\mathrm{size}; \\
&\mathrm{own} := \lambda t, \overline{v}.\, \exists \overline{\overline{v}}.\, \overline{v} = \sum_i \overline{\overline{v}}_i * \mathop{\Huge\ast}_i \llbracket \overline{\tau}_i \rrbracket_\gamma.\mathrm{own}(t, \overline{\overline{v}_i}); \\
&\mathrm{shr} := \lambda \kappa, t, \ell.\, \mathop{\Huge\ast}_i \llbracket \overline{\tau}_i \rrbracket_\gamma.\mathrm{shr}(\kappa, t, \ell + \textstyle\sum_{j<i} \llbracket \overline{\tau}_j \rrbracket_\gamma.\mathrm{size})\} \\
& && (\textsc{Ty-def-prod})
\end{aligned}$$

$$\begin{aligned}
\llbracket \Sigma \overline{\tau} \rrbracket_\gamma := \mathrm{Type}\,\{&\mathrm{size} := 1 + \max_i \llbracket \tau_i \rrbracket_\gamma.\mathrm{size}; \\
&\mathrm{own} := \lambda t, \overline{v}.\, \exists i, \overline{v}', \overline{v}''.\, \overline{v} = [i] \mathbin{+\!\!+} \overline{v}' \mathbin{+\!\!+} \overline{v}'' * \llbracket \overline{\tau}_i \rrbracket_\gamma.\mathrm{own}(t, \overline{v}') * \\
&\qquad |\overline{v}''| = 1 + \max_j \llbracket \overline{\tau}_j \rrbracket_\gamma.\mathrm{size}; \\
&\mathrm{shr} := \lambda \kappa, t, \ell.\, \exists i.\, \llbracket \overline{\tau}_i \rrbracket_\gamma.\mathrm{shr}(\kappa, t, \ell + 1) * \&^{\kappa}_{\mathbf{frac}}\big(\lambda q.\, \ell \xmapsto{q} i * \\
&\qquad (\ell + 1 + \llbracket \overline{\tau}_i \rrbracket_\gamma.\mathrm{size} \mapsto \lambda \overline{v}.\, \llbracket \overline{\tau}_i \rrbracket_\gamma.\mathrm{size} + |\overline{v}| = \max_j \llbracket \overline{\tau}_j \rrbracket_\gamma.\mathrm{size})\big)\} \\
& && (\textsc{Ty-def-sum})
\end{aligned}$$

$$\begin{aligned}
\llbracket \lightning\, 1 \rrbracket_\gamma &:= \mathrm{SimpleType}(\lambda\_,\_.\, \mathsf{True}) && (\textsc{Ty-def-uninit1}) \\
\llbracket \lightning\, n \rrbracket_\gamma &:= \Pi[\lightning\, 1, \ldots, \lightning\, 1] \qquad \text{of length } \llbracket n \rrbracket && (\textsc{Ty-def-uninit})
\end{aligned}$$

$$\begin{aligned}
\llbracket \forall \overline{\alpha}.\, \textsf{\textbf{fn}}(\digamma : \mathbf{E}; \overline{\tau}) \to \tau \rrbracket_\gamma := \mathrm{SimpleType}(&\lambda\_, v.\, \exists f, \overline{x}, k, F.\, v = \texttt{\textbf{funrec}}\, f(\overline{x})\, \texttt{\textbf{ret}}\, k := F * \\
&\triangleright \forall \overline{\kappa}, \kappa_\digamma, v_k, \overline{v}.\, \Box \llbracket \mathbf{E}; \digamma \sqsubseteq_1 [] \mid k \lhd \textsf{\textbf{cont}}(\digamma \sqsubseteq_1 []; x.\, x \lhd \textsf{\textbf{own}}\, \tau); \\
&\qquad \overline{x} \mathrel{\overline{\lhd}} \textsf{\textbf{own}}\, \overline{\tau} \vdash F[\texttt{\textbf{funrec}}\, f(\overline{x})\, \texttt{\textbf{ret}}\, k := F/f, \overline{v}/\overline{x}, v_k/k] \rrbracket_{\gamma[\overline{\alpha} \leftarrow \overline{\kappa}][\digamma \leftarrow \kappa_\digamma][\overline{x} \leftarrow \overline{v}][k \leftarrow v_k]}) \\
& && (\textsc{Ty-def-fn})
\end{aligned}$$

$$\llbracket \mu\, T.\, \tau \rrbracket := \mathit{fix}(\lambda T.\, \llbracket \tau \rrbracket_{\gamma[T \leftarrow T]})$$

Figure 7: The interpretations of all the primitive types

Simple sharing. The most simple form of sharing occurs when sharing "plain data". For example, consider the type **int**. We define ownership and sharing of a number as follows:

$$\begin{aligned}\llbracket \textsf{int} \rrbracket.\mathrm{own}(t, \overline{v}) &:= \exists z.\, \overline{v} = [z] \\ \llbracket \textsf{int} \rrbracket.\mathrm{shr}(\kappa, \_, \ell) &:= \exists \overline{v}.\ \&^{\kappa}_{\mathbf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} \overline{v}) * \llbracket \textsf{int} \rrbracket.\mathrm{own}(t, \overline{v})\end{aligned}$$

Actually, this kind of sharing is so common that we define a notion of *simple types* using the above sharing predicate:

$$\begin{aligned}\mathrm{SimpleType}(\Phi) := \mathrm{Type}\,\{&\mathrm{size} := 1; \\ &\mathrm{own} := \lambda t, \overline{v}.\, \exists v.\, \overline{v} = [v] * \Phi(t, v); \\ &\mathrm{shr} := \lambda \kappa, t, \ell.\, \exists v.\ \&^{\kappa}_{\mathbf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} v) * \triangleright \Phi(t, v)\}\end{aligned}$$

For a simple type, we only have to show that the ownership predicate Φ is persistent and that Ty-size holds. The remaining properties follow.

Sharing owned pointers and mutable references. It turns out that owned pointers and mutable references have interesting and similar sharing predicates. Here's the full definition:

$$\begin{aligned}\llbracket \textsf{own}_n\, \tau \rrbracket.\mathrm{shr}(\kappa, t, \ell) &:= \exists \ell'.\ \&^{\kappa}_{\mathbf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} \ell') * \Box\big(\forall q.\, [\kappa]_q \Rrightarrow\!\!\ast^{\mathcal{N}_{\mathrm{lft}}}_{\mathcal{N}_{\mathrm{shr}}, \mathcal{N}_{\mathrm{lft}}} \llbracket \tau \rrbracket.\mathrm{shr}(\kappa, t, \ell') * [\kappa]_q\big) \\ \llbracket \&^{\kappa}_{\textsf{mut}}\, \tau \rrbracket.\mathrm{shr}(\kappa', t, \ell) &:= \exists \ell'.\ \&^{\kappa'}_{\mathbf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} \ell') * \Box\big(\forall q.\, [\kappa \sqcap \kappa']_q \Rrightarrow\!\!\ast^{\mathcal{N}_{\mathrm{lft}}}_{\mathcal{E}, \mathcal{N}_{\mathrm{lft}}} \llbracket \tau \rrbracket.\mathrm{shr}(\kappa \sqcap \kappa', t, \ell') * [\kappa \sqcap \kappa']_q\big)\end{aligned}$$

using again the "magic update that takes a step" introduced in §4.

As you can see, a shared reference to an owned pointer or mutable reference consists of two parts: First of all, the outer pointer is shared. This is a straight-forward fractured borrow such that everybody gets a read-only permission to dereference the outer pointer. This is just like the simple sharing predicate defined above.

The more complicated, second part involves sharing the inner pointer. What we would like to have here (instead of the view shift that takes a step) is just $\llbracket \tau \rrbracket.\mathrm{shr}(\kappa, t, \ell')$. Unfortunately, that would make it impossible to prove Ty-share: Since owned pointers are, well, pointers, they only own the data they point to *later* or under a borrow, respectively. This also means they cannot start sharing that data immediately, since execution view shifts under $\triangleright$ is not possible. Instead, we remember that we can start sharing the data behind the inner pointer *once someone lets us take a step*. In other words, only when the outer pointer is dereferenced, the sharing of the inner data actually starts. We kind of lazily apply sharing down the pointer chain. To prove this view shift, there will be a small invariant managing this, since the shared references could actually be distributed accross different threads – there is a race for who gets to actually start the sharing.

5.1.3 Compound types: Sums and products

Both ownership and sharing of sums and products are fairly straight-forward, but there is some book-keeping and list manipulation going on that blurs the view.

$$\begin{aligned}
\llbracket \Pi\overline{\tau} \rrbracket.\mathrm{own}(t, \overline{v}) &:= \exists \overline{\overline{v}}.\, \overline{v} = \sum_i \overline{\overline{v}}_i * \mathop{\Large *}_{i} \llbracket \overline{\tau}_i \rrbracket.\mathrm{own}(t, \overline{v_i}) \\
\llbracket \Pi\overline{\tau} \rrbracket.\mathrm{shr}(\kappa, t, \ell) &:= \mathop{\Large *}_{i} \llbracket \overline{\tau}_i \rrbracket.\mathrm{shr}(\kappa, t, \ell + \textstyle\sum_{j<i} \llbracket \overline{\tau}_j \rrbracket.\mathrm{size}) \\
\llbracket \Sigma\overline{\tau} \rrbracket.\mathrm{own}(t, \overline{v}) &:= \exists i, \overline{v}', \overline{v}''.\, \overline{v} = [i] \mathbin{+\!\!+} \overline{v}' \mathbin{+\!\!+} \overline{v}'' * |\overline{v}''| = 1 + \max_j \llbracket \overline{\tau}_j \rrbracket.\mathrm{size} * \llbracket \overline{\tau}_i \rrbracket.\mathrm{own}(t, \overline{v}') \\
\llbracket \Sigma\overline{\tau} \rrbracket.\mathrm{shr}(\kappa, t, \ell) &:= \exists i.\, \&^{\kappa}_{\mathbf{frac}} \big(\lambda q.\, \ell \overset{q}{\mapsto} i * (\ell + 1 + \llbracket \overline{\tau}_i \rrbracket.\mathrm{size} \mapsto \lambda \overline{v}.\, \llbracket \overline{\tau}_i \rrbracket.\mathrm{size} + |\overline{v}| = \max_j \llbracket \overline{\tau}_j \rrbracket.\mathrm{size})\big) * \\
&\qquad \llbracket \overline{\tau}_i \rrbracket.\mathrm{shr}(\kappa, t, \ell + 1)
\end{aligned}$$

5.1.4 Copy, Send, Sync

$$\begin{aligned}
\llbracket \tau \ \mathsf{copy} \rrbracket_\gamma &:= (\Box \forall t, \overline{v}.\, \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t, \overline{v}) \Rightarrow \Box \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t, \overline{v})) \land \\
&\qquad \Box \forall \kappa, t, \ell, q.\, \llbracket \tau \rrbracket_\gamma.\mathrm{shr}(\kappa, t, \ell) \mathrel{-\!\!*} \langle [\mathrm{Na} : t.\mathcal{N}_{\mathrm{shr}}.[{\geq}\ell, {<}\ell + 1 + \llbracket \tau \rrbracket_\gamma.\mathrm{size}]] * [\kappa]_q \Longleftrightarrow \\
&\qquad\quad q'.\, [\mathrm{Na} : t.\mathcal{N}_{\mathrm{shr}}.(\ell + \llbracket \tau \rrbracket_\gamma.\mathrm{size})] * \triangleright \ell \xmapsto{q'} \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t) \rangle_{\mathcal{N}_{\mathrm{shr}}, \mathcal{N}_{\mathrm{lft}}} \\
\llbracket \tau \ \mathsf{send} \rrbracket_\gamma &:= \Box \forall t_1, t_2, \overline{v}.\, \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t_1, \overline{v}) \Rightarrow \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t_2, \overline{v}) \\
\llbracket \tau \ \mathsf{sync} \rrbracket_\gamma &:= \Box \forall \kappa, t_1, t_2, \ell.\, \llbracket \tau \rrbracket_\gamma.\mathrm{shr}(\kappa, t_1, \ell) \Rightarrow \llbracket \tau \rrbracket_\gamma.\mathrm{shr}(\kappa, t_2, \ell)
\end{aligned}$$

5.2 Type and continuation contexts

$$\begin{aligned}
\llbracket \mathbf{T} \rrbracket_\gamma &: \mathit{TId} \to \mathit{iProp} \\
\llbracket \emptyset \rrbracket_\gamma(t) &:= \mathsf{True} \\
\llbracket \mathbf{T}, p \lhd \tau \rrbracket_\gamma(t) &:= \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t, [\llbracket p \rrbracket]) * \llbracket \mathbf{T} \rrbracket_\gamma(t) \\
\llbracket \mathbf{T}, p \lhd^{\dagger\kappa} \tau \rrbracket_\gamma(t) &:= \big([\dagger \llbracket \kappa \rrbracket] \Rrightarrow\!\!\ast_{\top} \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t, [\llbracket p \rrbracket])\big) * \llbracket \mathbf{T} \rrbracket_\gamma(t)
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathbf{K} \rrbracket_\gamma &: \mathit{TId} \to \mathit{iProp} \\
\llbracket \emptyset \rrbracket_\gamma(t) &:= \mathsf{True} \\
\llbracket \mathbf{K}, k \lhd \textsf{\textbf{cont}}(\mathbf{L}; \overline{x}.\, \mathbf{T}) \rrbracket_\gamma(t) &:= \big(\forall \overline{v}.\, [\mathrm{Na} : t] * \llbracket \mathbf{L} \rrbracket_\gamma(1) * \llbracket \mathbf{T} \rrbracket_{\gamma[\overline{x} \leftarrow \overline{v}]}(t) \mathrel{-\!\!*} \mathsf{wp}\ \llbracket k \rrbracket_\gamma(\overline{v})\ \{\mathsf{True}\}\big) \land \llbracket \mathbf{K} \rrbracket_\gamma(t)
\end{aligned}$$

5.3 Lifetime contexts and judgments

The local and external lifetime contexts are interpreted as follows:

$$\begin{aligned}
\llbracket \mathbf{E} \rrbracket_\gamma &: \mathit{iProp} \\
\llbracket \emptyset \rrbracket_\gamma &:= \mathsf{True} \\
\llbracket \mathbf{E}, \kappa \sqsubseteq_{\mathrm{e}} \kappa' \rrbracket_\gamma &:= \llbracket \kappa \rrbracket_\gamma \sqsubseteq \llbracket \kappa' \rrbracket_\gamma * \llbracket \mathbf{E} \rrbracket_\gamma
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathbf{L} \rrbracket_\gamma &: \mathit{Fract} \to \mathit{iProp} \\
\llbracket \emptyset \rrbracket_\gamma(q) &:= \mathsf{True} \\
\llbracket \mathbf{L}, \kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} \rrbracket_\gamma(q) &:= \exists \kappa'.\, \llbracket \kappa \rrbracket_\gamma = \kappa' \sqcap (\sqcap \llbracket \overline{\kappa} \rrbracket_\gamma) * [\kappa']_q * \Box\big([\kappa']_1 \mathrel{\Rrightarrow\!\!\!*}^{\mathcal{N}_{\mathrm{lft}}}_{\emptyset} [\dagger \kappa']\big) * \llbracket \mathbf{L} \rrbracket_\gamma(q)
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathbf{E}_1 ; \mathbf{L}_1 \vdash \mathbf{E}_2 \rrbracket &:= \forall q.\, \llbracket \mathbf{L} \rrbracket(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E}_1 \rrbracket \mathrel{-\!\!*} \llbracket \mathbf{E}_2 \rrbracket) \\
\llbracket \mathbf{E} ; \mathbf{L} \vdash \kappa_1 \sqsubseteq \kappa_2 \rrbracket &:= \forall q.\, \llbracket \mathbf{L} \rrbracket(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \llbracket \kappa_1 \rrbracket \sqsubseteq \llbracket \kappa_2 \rrbracket) \\
\llbracket \mathbf{E} ; \mathbf{L} \vdash \kappa\ \mathsf{alive} \rrbracket &:= \forall q.\, \llbracket \mathbf{E} \rrbracket \mathrel{-\!\!*} \langle \llbracket \mathbf{L} \rrbracket(q) \Longleftrightarrow q'.\, [\llbracket \kappa \rrbracket]_{q'} \rangle
\end{aligned}$$

5.4 Judgments

$$\begin{aligned}
\llbracket \mathbf{E} ; \mathbf{L} \vdash \tau_1 \Rightarrow \tau_2 \rrbracket_\gamma &:= \forall q.\, \llbracket \mathbf{L} \rrbracket_\gamma(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E} \rrbracket_\gamma \mathrel{-\!\!*} \tau_1 \sqsubseteq^{\mathsf{ty}} \tau_2) \\
\llbracket \mathbf{E} ; \mathbf{L} \vdash \mathbf{T}_1 \Rightarrow \mathbf{T}_2 \rrbracket_\gamma &:= \forall t, q.\, \llbracket \mathbf{E} \rrbracket_\gamma * \llbracket \mathbf{L} \rrbracket_\gamma(q) * \llbracket \mathbf{T}_1 \rrbracket_\gamma(t) \mathrel{\Rrightarrow\!\!\!*} \llbracket \mathbf{L} \rrbracket_\gamma(q) * \llbracket \mathbf{T}_2 \rrbracket_\gamma(t) \\
\llbracket \mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2 \rrbracket_\gamma &:= \forall t.\, [\dagger\kappa] * \llbracket \mathbf{T}_1 \rrbracket_\gamma(t) \mathrel{\Rrightarrow\!\!\!*} \llbracket \mathbf{T}_2 \rrbracket_\gamma(t) \\
\llbracket \mathbf{E} \vdash \mathbf{K}_1 \Rightarrow \mathbf{K}_2 \rrbracket_\gamma &:= \forall t.\, \llbracket \mathbf{E} \rrbracket_\gamma * \llbracket \mathbf{K}_1 \rrbracket_\gamma(t) \mathrel{\Rrightarrow\!\!\!*} \llbracket \mathbf{K}_2 \rrbracket_\gamma(t) \\
\llbracket \mathbf{E} ; \mathbf{L} \vdash \tau_1 \multimap^{\tau} \tau_2 \rrbracket_\gamma &:= \forall \ell, t, q.\, \llbracket \mathbf{E} \rrbracket_\gamma * \llbracket \mathbf{L} \rrbracket_\gamma(q) * \llbracket \tau_1 \rrbracket_\gamma.\mathrm{own}(t, [v]) \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathrm{lft}}, \mathcal{N}_{\mathrm{rust}}} \exists \ell, \overline{v}.\, \ell = v * |\overline{v}| = \llbracket \tau \rrbracket_\gamma.\mathrm{size} * \ell \mapsto \overline{v} * \\
&\qquad \Big(\triangleright \ell \mapsto \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t) \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathrm{lft}}, \mathcal{N}_{\mathrm{rust}}} \llbracket \mathbf{L} \rrbracket_\gamma(q) * \llbracket \tau_2 \rrbracket_\gamma.\mathrm{own}(t, v) \Big) \\
\llbracket \mathbf{E} ; \mathbf{L} \vdash \tau_1 \multimapinv^{\tau} \tau_2 \rrbracket_\gamma &:= \forall \ell, t, q.\, \llbracket \mathbf{E} \rrbracket_\gamma * \llbracket \mathbf{L} \rrbracket_\gamma(q) * [\mathrm{Na} : t] * \llbracket \tau_1 \rrbracket_\gamma.\mathrm{own}(t, [v]) \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathrm{lft}}, \mathcal{N}_{\mathrm{rust}}} \\
&\qquad \exists \ell, \overline{v}, q'.\, \ell = v * \ell \xmapsto{q'} \overline{v} * \triangleright \llbracket \tau \rrbracket_\gamma.\mathrm{own}(t, \overline{v}) * \\
&\qquad \Bigg(\ell \xmapsto{q'} \overline{v} \mathrel{\Rrightarrow\!\!\!*}_{\mathcal{N}_{\mathrm{lft}}, \mathcal{N}_{\mathrm{rust}}} \llbracket \mathbf{L} \rrbracket_\gamma(q) * [\mathrm{Na} : t] * \llbracket \tau_2 \rrbracket_\gamma.\mathrm{own}(t, v) \Bigg) \\
\llbracket \mathbf{E} ; \mathbf{L} \mid \mathbf{T}_1 \vdash I \dashv x.\, \mathbf{T}_2 \rrbracket_\gamma &:= \forall t.\, \llbracket \mathbf{E} \rrbracket_\gamma * \llbracket \mathbf{L} \rrbracket_\gamma(1) * [\mathrm{Na} : t] * \llbracket \mathbf{T}_1 \rrbracket_\gamma(t) \mathrel{-\!\!*} \mathsf{wp}\, I \left\{ v.\, \llbracket \mathbf{L} \rrbracket_\gamma(1) * [\mathrm{Na} : t] * \llbracket \mathbf{T}_2 \rrbracket_{\gamma[x \leftarrow v]}(t) \right\} \\
\llbracket \mathbf{E} ; \mathbf{L} \mid \mathbf{K} ; \mathbf{T} \vdash F \rrbracket_\gamma &:= \forall t.\, \llbracket \mathbf{E} \rrbracket_\gamma * \llbracket \mathbf{L} \rrbracket_\gamma(1) * [\mathrm{Na} : t] * \llbracket \mathbf{K} \rrbracket_\gamma(t) * \llbracket \mathbf{T} \rrbracket_\gamma(t) \mathrel{-\!\!*} \mathsf{wp}\, F\, \{\mathsf{True}\}
\end{aligned}$$

5.5 Theorems

Theorem 1 (Compatibility of the logical relation)**.** *For any inference rule of the type system, if we wrap all judgments in semantic brackets $\llbracket - \rrbracket$, the resulting Iris theorem holds.*

Theorem 2 (Adequacy of the logical relation)**.** *Let f be a λ_{Rust} function such that the Iris assertion $\llbracket \emptyset ; \emptyset \mid \emptyset \vdash f \dashv x.\, x \lhd \mathbf{fn}() \to \Pi[] \rrbracket$ holds, then when we execute $f(\lambda x.\, x)$ (passing it the default continuation), no execution ends in a stuck state.*