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

Ralf Jung  Jacques-Henri Jourdan  Robbert Krebbers  Derek Dreyer

November 9, 2017

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Syntax</td>
<td>2</td>
</tr>
<tr>
<td>1.1</td>
<td>Grammar</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Operational semantics</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Continuation-passing-style let-normal programs</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Type System</td>
<td>6</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Well-formedness</td>
<td>7</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Size, Copy, Send, Sync</td>
<td>8</td>
</tr>
<tr>
<td>1.4.3</td>
<td>Lifetime context judgments</td>
<td>9</td>
</tr>
<tr>
<td>1.4.4</td>
<td>Type Inclusion</td>
<td>9</td>
</tr>
<tr>
<td>1.4.5</td>
<td>Well-typed functions and steps</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Some examples</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>$\lambda_{\text{Rust}}$ in Iris</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>Lifetime logic</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Proof rules</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Derived forms of borrowing</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Model</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>$\lambda_{\text{Rust}}$ model</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>Types</td>
<td>34</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Owned pointers and mutable references</td>
<td>35</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Shared references</td>
<td>35</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Compound types: Sums and products</td>
<td>38</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Copy, Send, Sync</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Type and continuation contexts</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Lifetime contexts and judgments</td>
<td>39</td>
</tr>
<tr>
<td>5.4</td>
<td>Judgments</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Theorems</td>
<td>39</td>
</tr>
</tbody>
</table>
1 Syntax

1.1 Grammar

$\lambda_{\text{Rust}}$ 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). $\ell$ is a heap location, their structure will be defined later. $x, f$ are all program variables, the second usually denoting a function. $\tau$ is a list of $x$ (and similar for other metavariables), which can be constructed as $[x_1, x_2, \ldots]$. 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 ($\text{na}$) or sequentially consistent ($\text{sc}$). 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 $\text{na}'$ 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:

$$\text{Mem} := \mathbb{N} \times \mathbb{N} \overset{\text{in}}{\mapsto} \text{LockSt} \times \text{Val}$$

The lock states encode a per-location reader-writer-lock that serves to detect data races. Notice that $\text{reading0}$ 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 $\text{O-cas-stuck}$ 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 $0()$.
The notation \[ h[\ell \leftarrow v] \] denotes the map \( h \) updated with location \( \ell \) to map to \( v \). \( h[\ell \leftarrow v | x \in T] \) does the update for every \( x \in T \), where \( \ell \) and \( v \) may depend on \( x \).

Note how \texttt{O-alloc} initializes memory with \( \ast \) (poison), modeling uninitialized memory. Similarly, instructions that don’t actually return any information (like storing) return \( \ast \), 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 \texttt{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 \texttt{return} or \texttt{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
\[
\begin{align*}
    & h \vdash z = z & h \vdash \ell = \ell & \ell_1 \notin \text{dom}(h) \lor \ell_2 \notin \text{dom}(h) & h \vdash \ell_1 = \ell_2 \\
    & \frac{z_1 \neq z_2}{h \vdash z_1 \neq z_2} & \frac{\ell_1 \neq \ell_2}{h \vdash \ell_1 \neq \ell_2} & h \vdash v_1 \neq v_2
\end{align*}
\]

\begin{align*}
\text{O-ectx} & \quad \frac{h \mid e \rightarrow h' \mid e_1', e_2'}{K[e] \rightarrow h \mid K[e_1'], e_2'} \\
\text{O-proj} & \quad \frac{h \mid \ell, n \rightarrow h \mid \ell + n}{h \mid \ell, n} \\
\text{O-assign-na} & \quad \frac{z_1 + z_2 = z'}{h \mid z_1 + z_2 \rightarrow h \mid z'} \\
\text{O-assign-na'} & \quad \frac{z_1 - z_2 = z'}{h \mid z_1 - z_2 \rightarrow h \mid z'} \\
\text{O-alloc} & \quad \begin{cases} n > 0 & i = (i', n') \{ i \} \times \mathbb{N} \# \text{dom}(h) \quad h' = h[\ell + m \leftarrow \langle \text{reading} 0, \star \rangle | m \in [n\ldots)] \\ h \mid \text{alloc}(n) \rightarrow h' \mid \ell \end{cases} \\
\text{O-free} & \quad \begin{cases} n > 0 & \ell = (i', n') \quad \text{dom}(h) \cap \{ i \} \times \mathbb{N} = \{ i \} \times ([n', <n' + n]) \quad h' = h[\ell + m \leftarrow \bot | m \in [n\ldots)] \\ (h \mid \text{free}(n, \ell)) \rightarrow (h' \mid \star) \end{cases} \\
\text{O-deref-sc} & \quad \frac{h(\ell) = \langle \text{reading} n, v \rangle}{h \mid \text{sc} \ell \rightarrow h \mid v} \\
\text{O-deref-na} & \quad \frac{h(\ell) = \langle \text{reading} n, v \rangle}{(h \mid \text{na} \ell) \rightarrow (h[\ell \leftarrow \langle \text{reading} n, v \rangle] \mid v)} \\
\text{O-deref-na'} & \quad \frac{h(\ell) = \langle \text{reading} n + 1, v \rangle}{(h \mid \text{na} \ell) \rightarrow (h[\ell \leftarrow \langle \text{reading} n, v \rangle] \mid v)} \\
\text{O-assign-sc} & \quad \frac{h(\ell) = \langle \text{reading} 0, v' \rangle}{h \mid \ell \leftarrow \text{sc} v \rightarrow (h[\ell \leftarrow \langle \text{reading} 0, v \rangle] \mid \star)} \\
\text{O-assign-na} & \quad \frac{h(\ell) = \langle \text{writing} v' \rangle}{(h \mid \ell \leftarrow \text{na} v) \rightarrow (h[\ell \leftarrow \langle \text{reading} 0, v \rangle] \mid \star)} \\
\text{O-assign-na'} & \quad \frac{h(\ell) = \langle \text{writing} v' \rangle}{(h \mid \ell \leftarrow \text{na} v) \rightarrow (h[\ell \leftarrow \langle \text{reading} 0, v \rangle] \mid \star)} \\
\text{O-cas-fail} & \quad \frac{h(\ell) = \langle \text{reading} n, v' \rangle}{(h \mid \text{CAS}(\ell, v_1, v_2)) \rightarrow (h \mid 0) \quad \ell \neq v_1} \\
\text{O-cas-suc} & \quad \frac{h(\ell) = \langle \text{reading} 0, v' \rangle}{(h \mid \text{CAS}(\ell, v_1, v_2)) \rightarrow (h \mid 0) \quad \ell \neq \ell_1} \\
\text{O-case} & \quad \frac{h(\ell) = \langle \text{reading} 0, v_1 \rangle}{h \mid \text{case of } \tau \rightarrow (h \mid \tau')} \\
\text{O-app} & \quad \frac{h(\text{rec } f(\tau) := e)(\tau)) \rightarrow (h \mid e[\text{rec } f(\tau) := e/f, \tau/\tau])}{h \mid \text{fork } \{ e \} \rightarrow h \mid \star, e} \\
\end{align*}

Figure 2: Operational semantics.
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.

\[
\begin{align*}
\text{funrec } f(\pi) \text{ ret } k & := e := \text{rec } ([k] + \pi) := e \\
\text{let } x = e \text{ in } e' & := (\text{rec }_x([x]) := e')(e) \\
\text{e' ; e := let } = = e' \text{ in } e \\
\text{letcont } k(\pi) & := e \text{ in } e' := \text{let } k = (\text{rec } k(\pi) := e) \text{ in } e' \\
\text{jump } k(\tau) & := k(\tau) \\
\text{call } f(\pi) \text{ ret } k & := f([k] + \pi) \\
\text{false} & := 0 \\
\text{true} & := 1 \\
\text{if } e_0 \text{ then } e_1 \text{ else } e_2 & := \text{case } e_0 \text{ of } [e_1, e_2]
\end{align*}
\]

\[
\begin{align*}
* e & := *_{\text{na}} e \\
\text{e}_1 & := e_2 := e_1 :=_{\text{na}} e_2 \\
\text{new} & := \text{rec } \text{new}(\text{size}) := \\
& \quad \text{if } \text{size} == 0 \text{ then } (42, 1337) \text{ else alloc}(\text{size}) \\
\text{delete} & := \text{rec } \text{delete}(\text{size}, \text{ptr}) := \\
& \quad \text{if } \text{size} == 0 \text{ then } \# \text{ else free}(\text{size}, \text{ptr}) \\
\text{memcpy} & := \text{rec } \text{memcpy}(\text{dst}, \text{len}, \text{src}) := \\
& \quad \text{if } \text{len} \leq 0 \text{ then } \# \text{ else}
\quad \text{dst}.0 := \text{src}.0; \\
& \quad \text{memcpy}(\text{dst.1}, \text{len} - 1, \text{src.1})
\end{align*}
\]

\[
\begin{align*}
\text{e}_1 :=_{\text{na}} * \text{e}_2 & := \text{memcpy}(\text{e}_1, n, \text{e}_2) \\
\text{e}_{\text{inj}_i} & := e.0 := i \\
\text{e}_1 :=_{\text{inj}_i} \text{e}_2 & := \text{e}_1.0 := i; \text{e}_1.1 := \text{e}_2 \\
\text{e}_1 :=_{*_{\text{inj}_i}} * \text{e}_2 & := \text{e}_1.0 := i; \text{e}_1.1 :=_{*} \text{e}_2
\end{align*}
\]

\[
\begin{align*}
\text{skip} & := \text{let } x = \# \text{ in } \# \\
\text{newlft} & := \# \\
\text{endlft} & := \text{skip}
\end{align*}
\]

We distinguish between three classes of expressions: function bodies \( F \) consist of instructions \( I \) that operate on paths \( p \). The \textbf{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 λRust, we are not concerned with whole programs, but only with the individual functions of a program.

\begin{align*}
\text{Path} & \ni p ::= x \mid p.n \\
\text{Instr} & \ni I ::= \text{false} \mid \text{true} \mid z \mid \text{funrec } f(\mathcal{\tau}) \text{ ret } k ::= F \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \leq p_2 \\
& \mid \text{new}(n) \mid \text{delete}(n, p) \mid ^*p \mid p_1 ::= p_2 \mid p \allowbreak ::= \mathcal{\tau} \mid p_1 ::= n \ast p_2 \mid p_1 ::= n \ast p_2 \\
\text{FuncBody} & \ni F ::= \text{let } x = I \text{ in } F \mid \text{letcont } k(\mathcal{\tau}) ::= F_1 \text{ in } F_2 \mid \text{endlift}; F \\
& \mid \text{if } p \text{ then } F_1 \text{ else } F_2 \mid \text{case } ^*p \text{ of } F \mid \text{jump } k(\mathcal{\tau}) \mid \text{call } f(\mathcal{\tau}) \text{ ret } k
\end{align*}

1.4 Type System

The key concept of the λRust 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 \( \Gamma \) contains all binders. It assigns variables to their sort \( \sigma \) (either program variable \( x : \text{val} \), a lifetime \( \alpha : \text{lft} \) or a type \( T : \text{type} \)). Furthermore, there is a context for external lifetimes \( E \), a context for local lifetimes \( L \), a context assigning types to variables \( T \) and a context managing continuations \( K \).

\begin{align*}
\text{Sort} & \ni \sigma ::= \text{val} \mid \text{lft} \mid \text{type} \\
\Gamma & ::= \emptyset \mid \Gamma, X : \sigma \\
\text{Lft} & \ni \kappa ::= \alpha \mid \text{static} \\
E & ::= \emptyset \mid E, \kappa \sqsubseteq \kappa' \\
L & ::= \emptyset \mid L, \kappa \sqsubseteq \kappa \\
\text{Mod} & \ni \mu ::= \text{mut} \mid \text{shr} \\
\text{Type} & \ni \tau ::= T \mid \text{bool} \mid \text{int} \mid \iota_n \\
& \mid \text{own}_n \tau \mid &_{\mu}^{\kappa} \tau \mid \Sigma \tau \mid \Pi \tau \mid \forall \tau. \text{fn}(\mathcal{\tau}) \to \tau \mid \mu T. \tau \\
T & ::= \emptyset \mid T, p < \tau \mid T, p <_{\kappa} \tau \\
K & ::= \emptyset \mid K, k < \text{cont}(L ; \tau, T)
\end{align*}

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 \text{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 \&_{\mu}^{\kappa} \tau, also called (temporarily) borrowed pointer. References are qualified by a modifier, which is either mut (mutable, i.e., unique) and 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 \( L \). 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 \((p ◄ \tau)\) and type assignments that are blocked by a lifetime: \(p ◄^{\kappa} \tau\) means that we can only use this type assignment again when \(\kappa\) has ended.

### 1.4.1 Well-formedness

The well-formedness judgments \((\vdash_{\text{wf}})\) 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 \(\Gamma\) implicit.

#### Well-formed paths

\[
\begin{align*}
x : \text{val} & \in \Gamma \\
\Gamma \vdash_{\text{wf}} x & \\
\Gamma \vdash_{\text{wf}} p \cdot n & \\
\end{align*}
\]

#### Well-formed lifetimes

\[
\begin{align*}
\alpha : \text{lft} & \in \Gamma \\
\Gamma \vdash_{\text{wf}} \alpha & \\
\end{align*}
\]

\[\Gamma \vdash_{\text{wf}} \text{static}\]

#### Well-formed external lifetime contexts

\[
\begin{align*}
\Gamma \vdash_{\text{wf}} E & \\
\Gamma \vdash_{\text{wf}} \emptyset & \\
\end{align*}
\]

#### Well-formed local lifetime contexts

\[
\begin{align*}
\Gamma \vdash_{\text{wf}} \emptyset & \\
\Gamma \vdash_{\text{wf}} L & \\
\forall \kappa' \in \kappa : \Gamma \vdash_{\text{wf}} \kappa' & \\
\Gamma \vdash_{\text{wf}} L', \kappa \sqsubseteq \kappa' & \\
\end{align*}
\]

#### Well-formed types

\[
\begin{align*}
T : \text{type} & \in \Gamma \\
\Gamma \vdash_{\text{wf}} T & \\
\Gamma \vdash_{\text{wf}} \text{bool} & \\
\Gamma \vdash_{\text{wf}} \text{int} & \\
\Gamma \vdash_{\text{wf}} \text{fn} & \\
\Gamma \vdash_{\text{wf}} \text{ownn} & \\
\Gamma \vdash_{\text{wf}} \text{cat} & \\
\forall i : \Gamma \vdash_{\text{wf}} \tau_i & \\
\forall i : \Gamma \vdash_{\text{wf}} \pi_i & \\
\Gamma, \pi, \tau : \Gamma \vdash_{\text{wf}} \text{lft} & \\
\forall i : \Gamma, \tau : \Gamma \vdash_{\text{wf}} \text{type} & \\
\Gamma \vdash_{\text{wf}} \forall \kappa \cdot \text{fn}(\kappa : \text{E}; \tau) \rightarrow \tau & \\
\Gamma, T : \text{type} \vdash_{\text{wf}} \tau & \\
\end{align*}
\]

\[T \text{ is guarded by pointer types in } \tau\]

#### Well-formed type contexts

\[
\begin{align*}
\Gamma \vdash_{\text{wf}} \emptyset & \\
\Gamma \vdash_{\text{wf}} T & \\
\Gamma \vdash_{\text{wf}} p \cdot \tau & \\
\Gamma \vdash_{\text{wf}} \text{type} & \\
\end{align*}
\]

\[\Gamma \vdash_{\text{wf}} T, p \triangleleft_{\kappa} \tau\]
Well-formed continuation contexts

$$\Gamma \vdash_{\text{wf}} K$$

$$\Gamma \vdash_{\text{wf}} \emptyset \quad \Gamma \vdash_{\text{wf}} T \quad \Gamma \vdash_{\text{wf}} k \quad \Gamma \vdash_{\text{wf}} L \quad \Gamma, \pi : \text{val} \vdash_{\text{wf}} T$$

$$\Gamma \vdash_{\text{wf}} T, k \leftarrow \text{cont}(L; \pi ; T)$$

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:

- \(\text{size}(\text{bool}) := 1\)
- \(\text{size}(\text{int}) := 1\)
- \(\text{size}(\iota \ n) := n\)
- \(\text{size}(\text{own}_n \ \tau) := 1\)
- \(\text{size}(\& \kappa \ \tau) := 1\)
- \(\text{size}(\Pi \ \tau) := \sum_i \text{size}(\tau_i)\)
- \(\text{size}(\Sigma \ \tau) := 1 + \max_i \text{size}(\tau_i)\)
- \(\text{size}(\mu T. \tau) := \text{size}(\tau)\)
- \(\text{size}(\forall \alpha. \text{fn}^{(f : E ; \tau)} \rightarrow \tau) := 1\)

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

\(\tau\) copy

- bool copy
- int copy
- \(\iota \ n\) copy
- \(\&_{\text{shr}} \ \tau\) copy
- \(\forall i. \tau_i\) copy
- \(\forall i. \tau_i\) copy
- \(\Pi \ \tau\) copy
- \(\Sigma \ \tau\) copy

\((\forall \alpha. \text{fn}^{(f : E ; \tau)} \rightarrow \tau)\) copy

\(T\) copy

\(\tau\) copy

\(\mu T. \tau\) 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

\(\tau\) send

- bool send
- int send
- \(\iota \ n\) send
- \(\text{own}_n \ \tau\) send
- \(\&_{\text{mut}} \ \tau\) send
- \(\&_{\text{shr}} \ \tau\) send
- \(\forall i. \tau_i\) send

\(\forall i. \tau_i\) send

\(\Sigma \ \tau\) send

\((\forall \alpha. \text{fn}^{(f : E ; \tau)} \rightarrow \tau)\) send

\(T\) send

\(\tau\) send

\(\mu T. \tau\) send
1.4.3 Lifetime context judgments

The following judgments express various properties of lifetime contexts.

**Lifetime inclusion**

\[ \Gamma \vdash E; L \vdash \kappa \subseteq \kappa' \]

\[ E; L \vdash \kappa \subseteq \kappa' \]

\[ E; L \vdash \kappa \subseteq \kappa' \]

\[ E; L \vdash \kappa \subseteq \kappa'' \]

\[ E; L \vdash \kappa \subseteq \kappa'' \]

\[ \Gamma \vdash E; L \vdash \kappa \text{ alive} \]

\[ E; L \vdash \text{static alive} \]

\[ E; L \vdash \kappa \text{ alive} \]

\[ E; L \vdash \kappa' \text{ alive} \]

\[ E; L \vdash \kappa \text{ alive} \]

\[ E; L \vdash \kappa' \text{ alive} \]

**Local lifetime context inclusion**

\[ \Gamma \vdash L_1 \Rightarrow L_2 \]

\[ L \Rightarrow L' \]

\[ L' \text{ is a permutation of } L \]

**External lifetime context satisfiability**

\[ \Gamma \vdash E_1; L_1 \vdash E_2 \]

\[ E_1; L_1 \vdash \emptyset \]

\[ E_1; L_1 \vdash \kappa \subseteq \kappa' \]

\[ E_1; L_1 \vdash \kappa \subseteq \kappa' \]

\[ E_1; L_1 \vdash E_2 \]

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 (\(\Rightarrow\)), which is meant as sugar for mutual inclusion.
Subtyping

\[
\begin{array}{ccc}
\text{T-refl} & \quad \quad & \text{T-trans} \\
E; L \vdash \tau & \Rightarrow & E; L \vdash \tau' \quad E; L \vdash \tau' \Rightarrow \tau'' \\
& \quad & E; L \vdash \tau \Rightarrow \tau'' \\
\text{T-uninit-prod} & \quad & \text{T-recc} \\
E; L \vdash \xi \Sigma \tau \Leftrightarrow \Pi_\Sigma \tau \\
\end{array}
\]

\[
\begin{array}{c}
\text{T-bor-lft} \\
E; L \vdash \kappa \sqsubseteq \kappa' \\
& \quad & E; L \vdash \kappa \sqsubseteq \kappa' \\
\end{array}
\]

\[
\begin{array}{c}
\text{T-bor-uninit-prod} \\
E; L \vdash \xi \Sigma \tau \Leftrightarrow \Pi_\Sigma \tau \\
\end{array}
\]

\[
\begin{array}{c}
\text{T-bor-mut} \\
E; L \vdash \tau_1 \leftrightarrow \tau_2 \\
E; L \vdash \&_{\mu \tau_1} \leftrightarrow \&_{\mu \tau_2} \\
\text{T-bor-shr} \\
E; L \vdash \tau_1 \Rightarrow \tau_2 \\
E; L \vdash \&_{\mu \tau_1} \Rightarrow \&_{\mu \tau_2} \\
\text{T-off} \\
E; L \vdash \mu \tau_1 \Rightarrow \mu \tau_2 \\
\end{array}
\]

\[
\begin{array}{c}
\text{T-sum} \\
\forall i. E; L \vdash \tau_i \Rightarrow \tau^i \\
E; L \vdash \Sigma \tau \Rightarrow \Sigma \tau^i \\
\end{array}
\]

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

\[
\begin{array}{c}
\text{T-frame} \\
\Gamma; \alpha; f : \text{lt} \mid E', E_0; L_0 \vdash E[\kappa/\alpha] \\
\Gamma; \alpha; f : \text{lt} \mid E', E_0; L_0 \vdash \tau \Rightarrow \tau' \\
\Gamma; \alpha; f : \text{lt} \mid E', E_0; L_0 \vdash \tau \Rightarrow \tau' \\
\Gamma; \alpha; f : \text{lt} \mid E', E_0; L_0 \vdash \tau \Rightarrow \tau' \\
\Gamma; \alpha; f : \text{lt} \mid E', E_0; L_0 \vdash \tau \Rightarrow \tau' \\
\end{array}
\]

The following judgment expresses that when \( \kappa \) ends, we can “unblock” the parts of the typing context that is blocked by \( \kappa \).
Type context unblocking

\[
\begin{array}{c}
\emptyset \Rightarrow \tau \quad T_1 \Rightarrow \tau \quad T_1 \Rightarrow \tau
\\
T_1, p < \tau \Rightarrow \tau, T_2, p < \tau
\\
T_1, p < \tau' \Rightarrow \tau', T_2, p < \tau'
\end{array}
\]

Continuation context inclusion

\[
\begin{array}{c}
\Gamma \mid E \triangleright K_1 \Rightarrow K_2
\\
K' \text{ is a permutation of } K
\\
\Gamma \mid E \Rightarrow K
\\
E \mid K, K' \Rightarrow K
\\
\Gamma, \tau : \text{val} \mid E ; L \Rightarrow T \Rightarrow T'
\\
\Gamma \mid E \mid K, k \triangleright \text{cont}(L ; \tau) \Rightarrow K', k \triangleright \text{cont}(L ; \tau', T')
\end{array}
\]

1.4.5 Well-typed functions and steps

Finally we come to the main typing judgment: \( \Gamma \mid E ; L \mid K : T \vdash F \) 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 \( \Gamma \mid E ; L \mid T_1 \vdash I \leftarrow x. T_2 \) features two typing contexts: if \( T_1 \) holds before the step is executed, then \( T_2 \) 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. \( E ; L \vdash \tau_1 \rightarrow \tau \tau_2 \) says that we can write something of type \( \tau \) to a location described by \( \tau_1 \), which will change the type of the location to \( \tau_2 \). Similarly, \( E ; L \vdash \tau_1 \leftarrow \tau \tau_2 \) says that when reading from a location of type \( \tau_1 \), we will read something of type \( \tau \) and the type of the locations changes to \( \tau_2 \).

Well-typed functions

\[
\begin{array}{c}
\text{F-CONSEQUENCE}
\\
L \Rightarrow L' \quad E ; L \mid T \Rightarrow T' \quad E \mid K \Rightarrow K' \quad E ; L' \mid K' ; T' \vdash F
\\
E ; L \mid K ; T \vdash F
\end{array}
\]

\[
\begin{array}{c}
\text{F-EQUALIZE}
\\
E, \alpha \subseteq \kappa, \kappa \subseteq E, \alpha ; L \mid K ; T \vdash F
\\
E ; L, \alpha \subseteq[\kappa] \mid K ; T \vdash F
\end{array}
\]

\[
\begin{array}{c}
\text{F-LET}
\\
\Gamma \mid E ; L \mid T_1 \vdash I \leftarrow x. T_2
\\
\Gamma, x : \text{val} \mid E ; L \mid K ; T_2, T \vdash F
\\
\Gamma \mid E ; L \mid K ; T \vdash \text{let } x = I \text{ in } F
\end{array}
\]
F-LETCONT
\[ \Gamma, k, \tau : \text{val} \quad \vdash E; L_1 . K, k \triangleleft \text{cont}(L_1; \tau).T'; T' \vdash F_1 \]
\[ \Gamma, k : \text{val} \quad \vdash E; L_2 . K, k \triangleleft \text{cont}(L_1; \tau).T'; T \vdash F_2 \]
\[ \Gamma \vdash E; L_2 . K; T \vdash \text{letcont}(k(\tau)) := F_1 \text{ in } F_2 \]

F-IF
\[ E; L \mid K; T \vdash F_1 \quad E; L \mid K; T \vdash F_2 \]
\[ \vdash E; L \mid K; T, p \text{ bool} \vdash \text{if } p \text{ then } F_1 \text{ else } F_2 \]

F-JUMP
\[ E; L \vdash T \Rightarrow T'(\eta/\tau) \]
\[ E; L \vdash k \text{ bool} \Rightarrow \text{cont}(L; \tau).T'; T \vdash \text{jump } k(\eta) \]

F-CALL
\[ E; L \vdash \text{call } f \text{ where } f : \text{lift} \mid E, f \sqsubseteq \kappa; L \vdash E' \]
\[ \Gamma \mid E; L \mid k \triangleleft \text{cont}(L; y; y \triangleleft \text{own } \tau).T'; T, f \triangleleft \text{fn}(E'; \tau) \Rightarrow \tau \Rightarrow \text{call } f(p) \Rightarrow \text{ret } k \]

F-NEWLFT
\[ T \vdash \text{newlft} \]
\[ \Gamma, \alpha : \text{lift} \mid E; L, \alpha \triangleright \kappa; K; T \vdash F \]
\[ \Gamma \vdash E; L \mid K; T \vdash \text{newlft}; F \]

F-ENDLFT
\[ \Gamma, \kappa \triangleright \kappa; E; L \mid K; T \vdash F \quad T \Rightarrow \kappa; T' \]
\[ \Gamma, \kappa \vdash \text{endlft} \]
\[ E; L \mid K; T \vdash F \]
\[ E; L \mid \text{endlft}; F \]

F-CASE-OWN
\[ \forall i. (E; L \mid K; T, p.0 \triangleleft \text{own } \tau, p) \vdash p \triangleleft \text{own } \tau. \tau_i, p.0 \triangleleft \text{own } \tau \vdash (E; L \mid K; T, p) \vdash \text{cont}(L; \tau; \tau_i - \text{size}(\tau_i)) \vdash F_i) \]
\[ \vdash (E; L \mid K; T, p) \vdash \text{case } p \text{ of } F \]

F-CASE-BOR
\[ \forall i. (E; L \mid K; T, p.1 \triangleleft \text{own } \tau, p) \vdash p \triangleleft \text{own } \tau. \tau_i, p.1 \triangleleft \text{cont}(L; \tau; \tau_i - \text{size}(\tau_i)) \vdash F_i) \]
\[ \vdash (E; L \mid K; T, p) \vdash \text{case } p \text{ of } F \]

Type writing

\[ \tau \vdash \text{size}(\tau) = \text{size}(\tau') \]
\[ \vdash \text{write-own } \tau \triangleleft \kappa; \tau' \Rightarrow \text{own } \tau \]

Type reading

\[ \tau \vdash \text{size}(\tau) = \text{size}(\tau) \]
\[ \vdash \text{read-own-copy } \tau \]
\[ \vdash \text{read-own-move } \tau \]
\[ \vdash \text{read-bor } \tau \]

Well-typed instructions

\[ \vdash \text{S-true } \]
\[ \vdash \text{S-false } \]
\[ \vdash \text{S-num } \]

\[ \vdash \text{I } \]

\[ \vdash \text{INT } \]

\[ \vdash \text{NUM } \]
\[\begin{align*}
\text{S-fn} & \quad \tau' \text{ copy} \quad \tau' \text{ send} \\
\Gamma, \pi, \varphi : \text{ft, f, x, k : val} \mid E, E' ; f \subseteq [\cdot] \mid k < \text{cont}(f \subseteq [\cdot]) ; y, y < \text{own} \tau ; \\
& \quad p \sqsubseteq \tau', x \sqsubseteq \text{own} \tau, f < \forall \pi. \text{fn}(f : E, \tau) \rightarrow \tau \vdash F \\
\text{S-path} & \quad \text{S-nat-op} \\
E ; L \mid p < \tau \vdash p + x. x < \tau & \quad E ; L \mid p_1 < \text{int}, p_2 < \text{int} \vdash p_1 \{+, -, \} p_2 \vdash x. x < \text{int} \\
\text{S-nat-leq} & \quad \text{S-assgn} \\
E ; L \mid p < \text{int}, p_2 < \text{int} \vdash p_1 \leq p_2 \vdash x. x < \text{bool} & \quad E ; L \mid \emptyset \vdash \text{new}(n) \vdash x. x < \text{own} \frac{n}{n} \\
\text{S-delete} & \quad \text{S-derif} \quad \text{S-new} \\
n = \text{size}(\tau) & \quad E ; L \mid \tau_1 \leftarrow^\tau \tau_1' \quad \text{size}(\tau) = 1 \\
E ; L \mid p < \text{own}, \tau \vdash \text{delete}(n, p) \vdash \emptyset & \quad E ; L \mid p < \tau_1 \vdash \ast p \vdash x. p < \tau_1', x < \tau \\
\text{S-derif-bor-own} & \quad \text{S-derif-bor-bor} \\
E ; L \mid \kappa \text{ alive} & \quad E ; L \mid \kappa \subseteq \kappa' \\
E ; L \mid p < \&^\mu \text{own}, \tau \vdash \ast p \vdash x. x < \&^\mu \tau & \quad E ; L \mid p < \&^\mu \&^\kappa \text{ mut} \tau \vdash \ast p \vdash x. x < \&^\mu \tau \\
\text{S-assgn} & \quad \text{S-sum-assgn-unit} \\
E ; L \mid \tau_1 \leftarrow \tau_1' & \quad \tau_i = \Pi[\cdot] \\
E ; L \mid p_1 \leq \tau_1, p_2 \leq \tau \vdash p_1 \leftarrow \tau_1' & \quad E ; L \mid \tau_1 \leftarrow \tau_1' \\
\text{S-sum-assgn} & \quad \text{S-sum-memcpy} \quad \text{S-sum-memcpy} \\
E ; L \mid \tau_1 = \tau & \quad \tau_1 = \tau \\
E ; L \mid p_1 \leq \tau_1, p_2 \leq \tau \vdash p_1 \leftarrow \tau_1' & \quad E ; L \mid \tau_2 \leftarrow \tau_2' \\
\text{size}(\tau) = n & \quad E ; L \mid \tau_1 \leftarrow \tau_1' \\
E ; L \mid p_1 \leq \tau_1, p_2 \leq \tau_2 \vdash p_1 \leftarrow \tau_1' & \quad E ; L \mid \tau_2 \leftarrow \tau_2' \\
\text{S-sum-memcpy} & \quad \tau_i = \tau \\
E ; L \mid p_1 \leq \tau_1, p_2 \leq \tau_2 \vdash p_1 \leftarrow \tau_1' & \quad E ; L \mid \tau_2 \leftarrow \tau_2' \quad \tau_i = \tau \\
\end{align*}\]
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 $\Pi_{\tau}$, $\{\}$ for $\Pi[\_\_\_\_\_]$, $\tau_1 + \tau_2 + \ldots$ for $\Sigma_{\tau}$ and $!$ for $\Sigma[\_\_\_]$. We use $\xi$ as sugar for $\xi$. Finally, own $\tau$ is short for own$_{\text{size}}(\tau)$.

In local lifetime contexts, we use $\alpha$ as notation for $\alpha \sqsubseteq \ell[\_]$.

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.

havecont $k$ in $F_1$ wherecont $k(\pi) := F_2 := \; \text{letcont } k(x) := F_2 \text{ in } F_1$

letcall $x = f(\overline{\alpha})$ in $F := \; \text{letcont } k(x) := F \text{ in call } f(\overline{\alpha}) \text{ ret } k$

letalloc $x : \tau := p$ in $F := \; \text{let } x = \text{new}(1) \text{ in } x := p \text{ in } F$

letalloc $x : \tau := \ast p$ in $F := \; \text{let } x = \text{new}(\tau) \text{ in } x := \ast p \text{ in } F$

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
\[
\Gamma, x : \text{val} | E; L | K; T, x < \tau \vdash F \quad E; L \vdash E' \\
\Gamma | E; L | K; T, f : \text{fn}(E', \tau) \rightarrow \tau, p < \tau \vdash \text{letcall } x = f(\overline{\alpha}) \text{ in } F
\]

F-letalloc-assgn
\[
\Gamma, x : \text{val} | E; L | K; T, x < \text{own } \tau \vdash F \\
\Gamma | E; L | K, T, p < \tau \vdash \text{letalloc } x : \tau := p \text{ in } F
\]

F-letalloc-memcpy
\[
\tau_1 \ast \tau_2 \quad \Gamma, x : \text{val} | E; L | K; T, x < \text{own } \tau, p < \tau_2 \vdash F \\
\Gamma | E; L | K; T, p < \tau_1 \vdash \text{letalloc } x : \tau := \ast p \text{ in } F
\]

Example 1: Mutable to shared reference, field reference.

1 struct Point { x: i32, y: i32 }
2 fn get_x<'a>(p: &'a mut Point) -> &'a i32 {
3 &(*p).x
4 }

The types translate as follows:

Point := int $\times$ int  
get_x := $\forall \alpha. \text{fn}(p : f \llc \alpha; \&_{\text{mut}} \text{Point}) \rightarrow \&_{\text{sh}} \text{int}$

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:

funrec get_x(p) ret ret :=
  let p' = *p in letalloc r : &$_{\text{sh}}$ int := p'.0 in  
delete(&$_{\text{mut}}$ Point, p); jump ret(r)
I will usually try to make the Rust code and the $\lambda_{\text{Rust}}$ 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: $\psi \subseteq \alpha$

\begin{align*}
\{ \text{ret} \ll \text{cont}(\psi; r \ll \text{own} & \&\alpha \text{int}; p \ll \text{own} & \&\alpha \text{Point} ) \\
\{ p \} & \text{let } p' = *p \in \{ p' \ll \&\alpha \text{mut Point}, p \ll \text{own } \} \\
\{ p' \} & \text{Split } \{ p'.0 \ll \&\alpha \text{mut int}, p'.1 \ll \&\alpha \text{int} \} \\
\{ p', r \} & \text{letalloc } r: &\&\alpha \text{int} := p'.0 \in \{ r \ll \text{own} & \&\alpha \text{int} \} \\
\{ p, \text{ret}, r \} & \text{delete}(\&\alpha \text{mut Point}, p); \text{ret}(r)
\end{align*}

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 $x \ll \tau$ for $\tau$ copy), the item will implicitly be duplicated. When specifying items of the form to be consumed, we will often shorten $X \ll -$ 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**

1. $\text{fn rebor}(\text{mut } t1: \text{Point}, \text{mut } t2: \text{Point}) \rightarrow \text{i32}$
2. $\text{let mut } x = &\text{mut } t1$;
3. $\text{let } y = &\text{mut } (*x).y$;
4. $x = &\text{mut } t2$;
5. $*y$
6. }

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.

$$\text{rebor} := \text{fn}(\text{Point}, \text{Point}) \rightarrow \text{int}$$
funrec rebor\( (t_1, t_2) \) \( \texttt{ret} \) \( \texttt{ret} \) :=
\begin{align*}
\text{newlft};
\text{letalloc } x \colon \&^\alpha \text{mut Point} := t_1; \\
\text{let } x' = \ast x \text{ in let } y = x'.0 \text{ in} \\
x := t_2 \text{ in} \\
\text{let } y' = \ast y \text{ in letalloc } r \colon \text{int} := y' \text{ in} \\
\text{endlft}; \text{delete} (\text{Point}, t_1); \text{delete} (\text{Point}, t_2); \text{delete} (\&^\alpha \text{mut Point}, x); \text{jump ret} (r)
\end{align*}

\{ \texttt{ret} \ll \text{cont}(\texttt{r}; \texttt{r} \ll \text{own \text{int}}); \texttt{t}_1 \ll \text{own \text{Point}}, \texttt{t}_2 \ll \text{own \text{Point}} \}
\text{newlft};
\text{Local lifetimes: } \alpha \sqsubseteq [ ]
\begin{align*}
\{ t_1, t_2 \} & \text{ Borrow at } \alpha \{ t_1 \ll \&^\alpha \text{mut Point}, t_1 \ll \&^\alpha \text{mut Point}, t_2 \ll \&^\alpha \text{mut Point}, t_2 \ll \&^\alpha \text{mut Point} \}
\{ t_1 \} & \text{ letalloc } x \colon \&^\alpha \text{mut Point} := t_1; \{ x \ll \text{own } \&^\alpha \text{mut Point} \}
\{ x \} & \text{ let } x' = \ast x \{ x' \ll \&^\alpha \text{mut Point}, x \ll \text{own } \frac{1}{\alpha} \}
\{ x', t_2 \} & x := t_2 \{ x \ll \text{own } \&^\alpha \text{mut Point} \}
\{ y \} & \text{ let } y' = \ast y \{ y' \ll \text{int}, y \}
\{ y' \} & \text{ letalloc } r \colon \text{int} := y' \{ r \ll \text{own } \text{int} \}
\{ t_1, t_2 \} \text{ endlt}; \{ t_1 \ll \text{own \text{Point}}, t_2 \ll \text{own \text{Point}} \}
\text{Local lifetimes: } \emptyset
\begin{align*}
\{ t_1, t_2, x, \texttt{ret}, r \} & \text{ delete} (\text{Point}, t_1); \text{delete} (\text{Point}, t_2); \text{delete} (\&^\alpha \text{mut Point}, x); \text{jump ret} (r)
\end{align*}

Example 3: Borrowing from a borrowed box.

Now we can consider this piece of Rust code:

\begin{verbatim}
1 fn unbox <'a>(b: &'a mut Box<Point>) -> &'a mut u32 {
2    let bx = &mut (*b).x;
3 } ...
\end{verbatim}

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. \( \lambda_{\text{Rust}} \) does not distinguish between stack and heap, so the two concepts unify: Box is just own .

\begin{align*}
\text{unbox} := \forall \alpha. \text{fn}(\texttt{f} : \texttt{f} \sqsubseteq \alpha; &^\alpha \text{mut \text{Point}}) \to &^\alpha \text{mut \text{int}}
\end{align*}

funrec unbox\( (b) \) \( \texttt{ret} \) \( \texttt{ret} \) :=
\begin{align*}
\text{let } b' = \ast b \text{ in let } bx = \ast b'. \text{in} \\
\text{letalloc } r \colon &^\alpha \text{mut \text{int}} := bx.0 \text{ in} \\
\text{delete} (\&^\alpha \text{mut Point}, b); \text{delete} (\text{Point}, bx); \text{jump ret} (r)
\end{align*}
Example 4: Struct initialization.

```racket
1 fn point (x: i32, y: i32) -> Point {
2  Point { x: x, y: y}
3 }
```

Example 5: Enum matching and initialization.

We assume the following meta-level type definition:

```
Option := λτ. () + τ
```

We will write `Option(τ)` for `τ` applied to `Option`.

```racket
1 fn option_as_mut<T>(o: &mut Option<T>) -> Option<&mut T> {
2   match o { 3 None => None,
4   Some(ref mut t) => Some(t)
5   }
6 }
```
option_as_mut is parametric over some type \( \tau \) on the meta-level.

\[
\text{option_as_mut}(\tau) := \forall \alpha. \text{fn}(\neg \in \alpha; \&^\alpha_{\text{mut}} \text{Option}(\tau)) \rightarrow \text{Option}(\&^\alpha_{\text{mut}} \tau)
\]

funrec option_as_mut(o) ret ret :=
let r = new(Option(\&^\alpha_{\text{mut}} \tau)) in
havecont k in
let o' = ^o in case ^o of
- r := (\text{inj0})
  jump k()
- r := o'.1;
  jump k()
wherecont k() :=
delete(\text{own} ^\alpha_{\text{mut}} \text{Option}(\tau), o); \text{jump ret}(r)

Context: \( \neg \in \alpha \)
{ret \cont{\neg \in \alpha; r \in \text{own} \text{Option}(\&^\alpha_{\text{mut}} \tau), o \in \text{own} ^\alpha_{\text{mut}} \text{Option}(\tau)}
{r \in \text{own} ^\alpha_{\text{mut}} \text{Option}(\tau), o \in \text{own} ^\alpha_{\text{mut}} \text{Option}(\tau)}

havecont k in
{r \cont{\neg \in \alpha; r \in \text{own} \text{Option}(\&^\alpha_{\text{mut}} \tau), o \in \text{own} ^\alpha_{\text{mut}} \text{Option}(\tau), r \in \text{own} ^\alpha_{\text{mut}} \text{Option}(\tau)}
{o} let o' = ^o in {o' \in \&^\alpha_{\text{mut}} \text{Option}(\tau), o \in \text{own} ^\alpha_{\text{mut}} \text{Option}(\tau)}
{o'} case ^o of
- {o'.1 \in \&^\alpha_{\text{mut}} (())}
  {r} r := (\text{inj0}) in {r \in \text{own} \text{Option}(\&^\alpha_{\text{mut}} \tau)}
  \{k, o, r\} \text{jump k}()
- {o'.1 \in \&^\alpha_{\text{mut}} \tau}
  {o'.1} r := o'.1 in {r \in \text{own} \text{Option}(\&^\alpha_{\text{mut}} \tau)}
  \{k, o, r\} \text{jump k}()
wherecont k() :=
{r \in \text{own} \text{Option}(\&^\alpha_{\text{mut}} \tau), o \in \text{own} ^\alpha_{\text{mut}} \text{Option}(\tau), o}; \text{jump ret}(r)

Example 6: Moving out of an enum.

1 \text{fn} \text{unwrap_or< ? >}(o: \text{Option< ? >}, \text{def}: ?) \rightarrow ? \{
2 \text{match o} \{
3 \quad \text{None} \Rightarrow \text{def},
4 \quad \text{Some}(t) \Rightarrow t
5 \}\n6 \}

unwrap_or is parametric over some type \( \tau \) on the meta-level.

\[
\text{unwrap_or} := \forall \alpha. \text{fn}(\neg \in \alpha; \text{Option}(\tau), \tau) \rightarrow \tau
\]
funrec unwrap_or(o, def) ret ret :=
    case "o of
    — delete(Option(τ), o); jump ret(def)
    — let alloc r: τ := o.lin
          delete(Option(τ), o); delete(τ, def); jump ret(r)

Example 7: Lazy lifetime initialization.

```rust
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 Two(κ) := &κ int × &κ int.

lazy_lft := fn() → ()

funrec lazy_lft() ret ret :=
    newlft;
    let t = new(2) in let f = new(1) in let g = new(1) in
    f := 42;
    t.0 := f; t.1 := f;
    let t0' = "t.0 in "t0';
    g := 23;
    t.1 := g;
    let r = new(0) in
    endlft; delete(2, t); delete(1, f); delete(1, g); jump ret(r)
```
\{ ret \triangleleft \text{cont}(f; r. r \triangleleft \text{own}()) \}

newlft:

Local lifetimes: $\alpha \sqsubseteq []$

{} let $t = \text{new}(2)$ in let $f = \text{new}(1)$ in let $g = \text{new}(1)$ in
{} $t \triangleleft \text{own} \{ 2, f \triangleleft \text{own} \{ 1, g \triangleleft \text{own} \{ 1 \} \}$
{} $f := 42; \{ f \triangleleft \text{own int} \}$
{} Borrow at $\alpha \{ f \triangleleft \alpha^\text{shr} \text{int}; f \triangleleft^{\alpha^1} \text{own int} \}$
{} $t.0 := f; t.1 := f; \{ t \triangleleft \text{own} \text{Two}(\alpha) \}$
{} $t \mapsto t\prime = \ast t.0 \text{ in } \{ t\prime \triangleleft^{\alpha^\alpha} \text{int}; t \}$
{} $t\prime \} \ast t\prime \cdot \{ t\prime \}$
{} $g := 23; \{ g \triangleleft \text{own int} \}$
{} Borrow at $\alpha \{ g \triangleleft \alpha^\text{shr} \text{int}; g \triangleleft^{\alpha^1} \text{own int} \}$
{} $g, t \cdot t.1 := g; \{ g, t \}$
{} let $r = \text{new}(0)$ in \{ $r \triangleleft \text{own}()$ \}
{} $f, g \cdot \text{endlft}; \{ g \triangleleft \text{own int}; f \triangleleft \text{own int} \}$
{} $t, f, g, ret \cdot \text{delete}(2, t); \text{delete}(1, f); \text{delete}(1, g); \text{jump ret}(r)$


3 \(\lambda_{\text{Rust in Iris}}\)

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 straightforward 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 \(\gamma_{\text{PhysVal}}\)), and their contents; and finite partial functions from blocks to (exclusive) tuples for start index and length of the block (instance name \(\gamma_{\text{PhysFree}}\)). For that to work out, we give a monoid structure to lock states with some unit \(\varepsilon\) and reading \(n \cdot \text{reading} m = \text{reading} n + m\).

\[
\text{PhVal} := \text{Loc} \text{Fin} \text{Frac}(\text{LockSt} \times \text{AG(Val)}) \\
\text{PhFree} := \mathbb{N} \text{Fin} \text{Frac}(\text{PowFin(\mathbb{N})})
\]

\[
\ell \xrightarrow{q} v := (\ell \xrightarrow{q} \text{reading}0,v,\text{AUTH(PhVal)}) \gamma_{\text{PhVal}} \\
\ell \mapsto 1 \mapsto v \\
\ell \xrightarrow{q,i} \tau := \bigotimes \ell + i \xrightarrow{q,i} \tau_i \\
\ell \mapsto \tau := \ell \mapsto \tau \\\n\n\n\]

\[
\downarrow^m \ell := \exists i, n, \ell = (i,n) \wedge \bigotimes i \xrightarrow{q,i} \text{reading}1,v,\text{AUTH(PhVal)} \gamma_{\text{PhysFree}} \\
\text{InvPhys} := \exists h, V, F, \text{Phy}(h) \cdot \bigotimes \text{Val} \gamma_{\text{PhysVal}} \cdot \bigotimes \text{PhFree} * \\
h = V \cdot (\forall i, \text{dom}(h) \cap \{i\} \times \mathbb{N} = F(i))
\]

where \(\downarrow^m \ell\) is the permission to deallocate \(\ell\) 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 \(\text{InvPhys} \mathcal{N}_{\text{Ph}}\) 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:

\[
\ell \xrightarrow{q} v \ast \ell \xrightarrow{q'} v' \Leftrightarrow \ell \xrightarrow{q+q'} v \ast v = v' \\
\downarrow^m \ell \ast \downarrow^m \ell + m \Leftrightarrow \downarrow^{m+m'} \ell
\]

\[
\{\text{True}\} \text{alloc}(n) \{\ell, \exists \tau, \ell \mapsto \tau \mapsto \text{reading}1, n = n \ast \downarrow^1 \ell \} \mathcal{N}_{\text{Ph}} \\
\{\ell \mapsto \tau \mapsto \text{reading}1, \text{free}(\tau), v\} \mathcal{N}_{\text{Ph}} \\
\{\ell \xrightarrow{q} v \ast \ell \xrightarrow{q} v' = v \ast \ell \xrightarrow{q} v'\} \mathcal{N}_{\text{Ph}} \\
\{\ell \xrightarrow{q} v \ast \ell \xrightarrow{q} v' = v \ast \ell \xrightarrow{q} v'\} \mathcal{N}_{\text{Ph}}
\]

\[
\ell \xrightarrow{0} w \{v', v = () \ast \ell \mapsto w\} \mathcal{N}_{\text{Ph}} \\
\{\ell \mapsto w \} \ell := w \{v', v = () \ast \ell \mapsto w\} \mathcal{N}_{\text{Ph}}
\]

\[
\mathcal{N}_{\text{Ph}} \mathcal{N}_{\text{Ph}}
\]

\[
\mathcal{N}_{\text{Ph}} \mathcal{N}_{\text{Ph}}
\]

\[
\mathcal{N}_{\text{Ph}} \mathcal{N}_{\text{Ph}}
\]

\[
\mathcal{N}_{\text{Ph}} \mathcal{N}_{\text{Ph}}
\]

\[
\mathcal{N}_{\text{Ph}} \mathcal{N}_{\text{Ph}}
\]

\[
\mathcal{N}_{\text{Ph}} \mathcal{N}_{\text{Ph}}
\]

\[
\mathcal{N}_{\text{Ph}} \mathcal{N}_{\text{Ph}}
\]

\[
\mathcal{N}_{\text{Ph}} \mathcal{N}_{\text{Ph}}
\]
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 $\lambda_{\text{Rust}}$ 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 $\kappa$”, written $\&_{\kappa} P$.

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 $\kappa$: 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 $\kappa$ is ongoing, so we need to be able to split this assertion: $[\kappa]_q$ denotes ownership of the fraction $q$ of $\kappa$. 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:

$$P \mathrel{\overset{\epsilon_2}{\rightsquigarrow}} Q := P \rightarrow \epsilon_1 \mid \epsilon_2 \triangleright \epsilon_2 \mid \epsilon_1 \triangleright Q$$

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 $\kappa$ (the full borrow), and ownership when $\kappa$ died (a view shift that lets us “inherit” $P$ from $\kappa$). 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:

$$\langle \&_{\kappa} P \ast [\kappa]_q \rightleftharpoons \triangleright P \rangle_{\text{Nlt}}$$

This lets us open full borrows ($\&_{\kappa} P$) 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 ($[\triangleright \kappa]$).
Figure 3: Lifetime logic assertions and proof rules

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
<th>Timeless</th>
<th>Persistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\kappa]_q$</td>
<td>Fraction $q$ of lifetime token for $\kappa$: Witnessing that the lifetime is still ongoing</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$[\triangleright \kappa]$</td>
<td>Witness confirming that the lifetime $\kappa$ is dead (i.e., it has ended)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>$&amp;_{\text{full}}^\kappa P$</td>
<td>Ownership of the full borrow of $P$ for $\kappa$</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>$&amp;_i^\kappa P$</td>
<td>There is an indexed borrow named $i$ of $P$ for $\kappa$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>$[\text{Bor} : i]_q$</td>
<td>Ownership of fraction $q$ of the indexed borrow $i$</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Lifetimes.** Lifetimes $\kappa$ form a cancellable PCM with intersection as the operation ($\cap$) and unit $\varepsilon$.

$$\kappa \subseteq \kappa' := (\forall q. ([\kappa]_q \equiv q' \cdot [\kappa']_q),_{\text{Nin}}) \ast ([\kappa'] \Rightarrow _{\text{Nin}} [\kappa])$$

**Lifetime creation and end.**

**Creating full borrows and using them.**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\triangleright P \Rightarrow <em>{\text{Nin}} &amp;</em>{\text{full}}^\kappa P \ast ([\triangleright \kappa] \Rightarrow _{\text{Nin}} \triangleright P)$</td>
<td>LFTL-borrow</td>
</tr>
<tr>
<td>$\triangleright \kappa \Rightarrow <em>{\text{Nin}} &amp;^\kappa</em>{\text{full}} P$</td>
<td>LFTL-bor-idx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&amp;_{\text{full}}^\kappa P \ast [\text{Bor} : i]_1$</td>
<td>Indexed borrows</td>
</tr>
</tbody>
</table>

**Indexed borrows.**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&amp;^\kappa_{\text{full}} P \Rightarrow \exists i, &amp;_i^\kappa P \ast [\text{Bor} : i]_1$</td>
<td>LFTL-bor-idx</td>
</tr>
<tr>
<td>$&amp;^\kappa_{\text{full}} P \Rightarrow \exists i, &amp;_i^\kappa P \ast [\text{Bor} : i]_1$</td>
<td>LFTL-bor-fract</td>
</tr>
<tr>
<td>$&amp;^\kappa_{\text{full}} P \Rightarrow \exists i, &amp;_i^\kappa P \ast [\text{Bor} : i]_1$</td>
<td>LFTL-bor-acc</td>
</tr>
<tr>
<td>$&amp;^\kappa_{\text{full}} P \Rightarrow \exists i, &amp;_i^\kappa P \ast [\text{Bor} : i]_1$</td>
<td>LFTL-bor-unnest</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&amp;^\kappa_{\text{full}} ([\text{Bor} : i]_1) \Rightarrow <em>{\text{Nin}} &amp;^\kappa</em>{\text{full}} P$</td>
<td>23</td>
</tr>
</tbody>
</table>
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:

\[
\langle \&_{\text{full}}^\kappa P \Leftrightarrow b \text{ if } b \text{ then } \triangleright P \text{ else } \lceil \kappa \rceil \rangle_{\text{Nin}}
\]

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 \(\kappa\) 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 \(\kappa\) is. Lifetimes \(\kappa\) form a partial commutative monoid with unit \(\varepsilon\). We will also refer to the composition operation \((\cap)\) 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:

\[
\kappa \sqsubseteq \kappa' := \square \left( \forall q. ([\kappa]_q \Leftrightarrow q'. [\kappa']_q)_{\text{Nin}} \right) \ast ([\kappa'] \Rightarrow \text{Nin} [\kappa])
\]

This says that \(\kappa\) is dynamically shorter than \(\kappa'\) if, given any fraction the token for \(\kappa\), we can produce some fraction of the token for \(\kappa'\). Furthermore, tokens showing that \(\kappa'\) has ended must be convertible to tokens showing that \(\kappa\) 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 \(\&_{\text{full}}^\kappa P\) 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 \(P \rightarrow \exists x \in \tau. P\). Then we run the closing view shift to finish the proof. In the other case, all we
get is $[[\kappa]]$. We run the closing view shift, and now we use the fact that $\tau$ is inhabited to obtain some $x$. We pick that $x$ as the witness for our goal, and then use \texttt{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 $\&^\kappa_{\text{full}} P$, but not exactly.

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

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

Both of these rules are derived from \texttt{LftL-idx-bor-unnest}.

\textbf{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
<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
<th>Timeless</th>
<th>Persistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&amp;^{\kappa/N}_{at}$ P</td>
<td>There is a <strong>atomic persistent borrow</strong> of $P$ for $\kappa$ in namespace $\mathcal{N}$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>$&amp;^{\kappa}_{frac}$ $\lambda q. P$</td>
<td>There is a <strong>fractured borrow</strong> of $\lambda q. P$ for $\kappa$</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>$&amp;^{\kappa/p\cdot\mathcal{N}}_{na}$ P</td>
<td>There is a <strong>non-atomic persistent borrow</strong> of $P$ for $\kappa$ in non-atomic invariant pool $p$, namespace $\mathcal{N}$</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Atomic persistent borrows**

\[
\text{LftL-Bor-at}\quad(N \not\# N_{init} \vdash \&^{\kappa/\mathcal{N}}_{full} P \Rightarrow N_{init} \&^{\kappa/\mathcal{N}}_{at} P)
\]

\[
\text{LftL-At-Acc-Atomic}\quad(\&^{\kappa/\mathcal{N}}_{at} P \vdash \langle \text{[True] } \iff b \text{ if } b \text{ then } P \text{ else } [\uparrow \kappa]\rangle_{N_{init}} \& \mathcal{N})
\]

\[
\text{LftL-At-Shorten}\quad(\kappa' \subseteq \kappa)\quad(\&^{\kappa/\mathcal{N}}_{at} P \Rightarrow \&^{\kappa'/\mathcal{N}}_{at} P)
\]

**Non-atomic persistent borrows**

\[
\text{LftL-Bor-na}\quad(\&^{\kappa}_{full} P \Rightarrow \mathcal{N} \square \&^{\kappa/p\cdot\mathcal{N}}_{na} P)
\]

\[
\text{LftL-Na-Acc}\quad(\&^{\kappa/p\cdot\mathcal{N}}_{na} P \vdash \langle [\kappa] \rangle_{q} \iff [Na:p\cdot\mathcal{N}] \iff P\rangle_{N_{init} \cdot \mathcal{N}}
\]

\[
\text{LftL-Na-Shorten}\quad(\kappa' \subseteq \kappa)\quad(N \subseteq N')\quad(\&^{\kappa/p\cdot\mathcal{N}}_{na} P \Rightarrow \&^{\kappa'/p\cdot\mathcal{N}}_{na} P)
\]

**Fractured borrows**

\[
\text{LftL-Bor-Fracture}\quad(\&^{\kappa}_{full} \Phi(1) \Rightarrow N_{init} \&^{\kappa}_{frac} \Phi)
\]

\[
\text{LftL-Frac-Acc}\quad(\forall q_{1}, q_{2}, \Phi(q_{1} \cdot q_{2}) \iff \Phi(q_{1}) \cdot \Phi(q_{2})\quad(\&^{\kappa}_{frac} \Phi \vdash \langle [\kappa] \rangle_{q} \iff q' \cdot \Phi(q') \rangle_{N_{init}})
\]

\[
\text{LftL-Frac-Acc-Atomic}\quad(\&^{\kappa}_{frac} \Phi \vdash \langle \text{[True] } \iff (b,q) \text{ if } b \text{ then } \Phi(q) \text{ else } [\uparrow \kappa]\rangle_{N_{init}})
\]

\[
\text{LftL-Frac-Shorten}\quad(\kappa' \subseteq \kappa)\quad(\&^{\kappa}_{frac} \Phi \Rightarrow \&^{\kappa'}_{frac} \Phi)
\]

Figure 5: Lifetime logic derived forms
be defined as follows:
\[ \&^{\kappa/N} P := \exists i. \&^{\kappa}_i P * (N \neq N_{\text{in}} * \exists q. \lbrack \text{Bor} : i \rbrack^N_{q} \lor N = N_{\text{in}} * \exists q. \lbrack \text{Bor} : i \rbrack^N_{q}) \]

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:
\[ \&^{\kappa/p.N_{\text{na}}} P := \exists i. \&^{\kappa}_i P * \text{Naln}^p.N_{\text{na}}([\text{Bor} : i]) \]

**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 \( \Phi \) 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 \( \Phi \) in the borrow.

The way this works is that we have an atomic persistent borrow which contains some fraction of \( \Phi \), 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 \( \text{LFTL-bor-acc-strong} \) witnesses this fact by providing \( [i\kappa] \) to the view shift that is applied.

Fractured borrows can be defined as follows:
\[ \&^{\kappa/\text{frac}}_\Phi := \exists \kappa', \gamma, \kappa \subseteq \kappa' * \&^{\kappa'/N_{\text{in}}} \exists q. \Phi(q) * \frac{1}{q} \gamma * (q = 1 \lor [\kappa']_{1-q}) \]

Here, are using the FRAC RA.

Fractured borrows are particularly interesting for giving rise to dynamic lifetime inclusion \( \text{(LFTL-fract-lincl)} \).

### 4.3 Model

We will model lifetimes \( \kappa \) 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:

- **BorSt**: \( \text{in} + \text{open}(q) + \text{rebor}(\kappa) \)
- **LftSt**: \( \text{LftStAlive} + \text{LftStDead} \)
- **ALft**: \( \text{Auth}(N \xrightarrow{\text{fin}} \text{FRAC} + i () ) \)
- **ILft**: \( \text{Auth}(q^{\text{fin}}.+(N) \xrightarrow{\text{fin}} \text{AG} (G \times G \times G) ) \)
- **BorBox**: \( \text{Auth}(N \xrightarrow{\text{fin}} \text{AG}(\text{BorSt}) \times \text{FRAC}) \)
- **Cnt**: \( \text{Auth}(N) \)
- **InhBox**: \( \text{Auth}(q^{\text{fin}}(N)) \)
We assume some globally known indices $\gamma_a$ and $\gamma_i$ for managing atomic and intersected lifetimes. The two tokens of the lifetime logic are easily modelled:

$$
\kappa_q := \bigstar \bigwedge_{\Lambda \in \kappa} \Lambda \leftarrow \text{inl}(q) \downarrow \gamma_a
$$

$$
\kappa^\dagger := \exists \Lambda \in \kappa. \circ \bigwedge \Lambda \leftarrow \text{inr}(q) \downarrow \gamma_a
$$

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

$$
P \Rightarrow \bigstar [P_F]_E Q := P * P_F \Rightarrow \bigstar_0 Q * P_F
$$

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:

$$
\text{OwnBor}(\kappa, x) := \exists \gamma_{\text{bor}}. \bigwedge_{\Lambda \in \kappa} \Lambda \leftarrow \gamma_{\text{bor}} \downarrow \gamma_i \ast \bigwedge \gamma_{\text{bor}}
$$

$$
\text{OwnCnt}(\kappa, x) := \exists \gamma_{\text{cnt}}. \bigwedge_{\Lambda \in \kappa} \Lambda \leftarrow \gamma_{\text{cnt}} \downarrow \gamma_i \ast \bigwedge \gamma_{\text{cnt}}
$$

$$
\text{OwnInh}(\kappa, x) := \exists \gamma_{\text{inh}}. \bigwedge_{\Lambda \in \kappa} \Lambda \leftarrow \gamma_{\text{inh}} \downarrow \gamma_i \ast \bigwedge \gamma_{\text{inh}}
$$

Now we can define the core of the model: the protocols for alive and dead lifetimes. We split the namespace $N_{\text{fin}}$ into three disjoint sub-namespaces $N_{\text{mgmt}}, N_{\text{bor}}, N_{\text{inh}}$. Here, $A : \mathbb{N} \overset{\text{fin}}{\rightarrow} \text{LftSt}$ is a map indicating the state of the atomic lifetimes, and $I : \gamma^{\text{fin,+}}(N) \overset{\text{fin}}{\rightarrow} \mathcal{G} \times \mathcal{G} \times \mathcal{G}$ indicates which intersected lifetimes exist.
Alternatively, the borrow can be reborrowed to a strictly shorter lifetime (in $LftVs$ lifetime ends, after which roughly speaking says that one can view shift from are all in the same state, but there is still an authoritative set that manages ownership of slices for the inheritance side, the box overall contains assertion

\[
\text{LftBorAlive}(\kappa, P_B) := \exists B : \mathbb{N} \xrightarrow{\text{fin}} \text{BorSt. OwnBor}(\kappa, \bullet [i \leftarrow (B(i), 1) | i \in \text{dom}(B)]) * \Box(\mathcal{N}_{\text{bor}}, P_B, [i \leftarrow \text{bor_to_box}(B(i)) | i \in \text{dom}(B)]) *
\]

\[
\begin{cases}
  \text{True} & \text{if } B(i) = \text{in} \\
  \kappa \circ q & \text{if } B(i) = \text{open}(q)
\end{cases}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{LftBorDead}(\kappa) := \exists B : \mathbb{N} \xrightarrow{\text{fin}} \text{BorSt. OwnBor}(\kappa, \bullet [i \leftarrow (B(i), 1) | i \in B]) * \\
\text{Box}(\mathcal{N}_{\text{bor}}, P_B, [i \leftarrow \text{empty} | i \in \text{dom}(B)])
\end{array} \\
\text{LftInh}(\kappa, P_I, s) := \exists E : \mathbb{N} \xrightarrow{\text{fin}} \text{OwnInh}(\kappa, \bullet E) * \text{Box}(\mathcal{N}_{\text{inh}}, P_I, [i \leftarrow s | i \in E])
\end{array}
\]

\[
\begin{array}{c}
\text{LftVs}(\kappa, P_B, P_I) := \exists n : \mathbb{N} \text{OwnCnt}(\kappa, \bullet n) \forall I : \mathbb{N} \xrightarrow{\text{fin}} \mathcal{G} \times \mathcal{G} \times \mathcal{G},
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{bor_to_box}(s) := \begin{cases}
  \text{full} & \text{if } s = \text{in} \\
  \text{empty} & \text{otherwise}
\end{cases}
\end{array}
\end{array}
\]

Notice that $\text{LftAlive}$ and $\text{LftVs}$ are defined mutually recursively, which is well-defined because the size of the lifetime $\kappa$ gets strictly smaller.

The rough idea behind this setup is as follows: For every lifetime $\kappa$, we have two boxes: one tracking the borrow, 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 $\text{LFTL.borrow}$ and $\text{LFTL-reborrow}$. 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 $P_I$. $\text{LftInh}$ 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 $\text{LftVs}$, 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 $\text{LftVs}$ is needed to support reborrowing.

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

\[
\begin{align*}
\text{LftInv} & := \exists A, I. \text{LftInv}^{\text{mgr}}(A, \kappa) \\
\text{LftLCtx} & := \text{LftInv}^{\text{mgr}}
\end{align*}
\]

Now we can model the remaining assertions of the logic, and an assertion \text{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 \(\kappa\) and a box index \(j\).

\[
\begin{align*}
\kappa \subseteq \kappa' & := (\forall q. (([\kappa]_q \Leftrightarrow q'. [\kappa']_q)_{\text{Nin}}) * ([\kappa'] \Rightarrow \text{Nin} \ [\kappa]) \\
[\text{Bor} : \kappa', i, j]_q & := \text{OwnBor}(\kappa, o i \leftarrow (\text{in}, q)) \\
\&^\kappa_{\kappa', j} P & := \kappa \subseteq \kappa' * \text{BoxSlice}(\text{Nbor}, P, i) \\
\text{RawBor}(\kappa, P) & := \exists i. \text{BoxSlice}(\text{Nbor}, P, i) * [\text{Bor} : \kappa, i]_1 \\
\&^\kappa_{\text{full}} P & := \exists \kappa'. \kappa \subseteq \kappa' \ast \text{RawBor}(\kappa', P)
\end{align*}
\]

Some proof rules are trivially justified: \text{LftL-tok-fract}, \text{LftL-tok-comp}, \text{LftL-tok-unit}, \text{LftL-not-own-end}, \text{LftL-end-comp}, \text{LftL-end-unit}, \text{LftL-bor-idx}, \text{LftL-bor-fract}, \text{LftL-idx-shorten} are all simple implications. We briefly discuss the most important steps of most the remaining rules.\footnote{The proof of reborrowing is not covered here; it can (like the others) be found in the Coq development.} When we have a full borrow \&^\kappa_{\text{full}} P, we will call the actual lifetime of the borrow (the one in the existential quantifier) \(\kappa_0\). In particular, \(\kappa \subseteq \kappa_0\).

\textbf{Proof sketch of LftL-borrow.} First we have to check whether \(\kappa\) is already allocated in \(I\); if not, that’s easy to do:

\[
\text{True} \Rightarrow \text{Nin} \ [\kappa] 
\]

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 LftInv\((A, \kappa)\). If the lifetime is dead, we use the following to create a “fake” full borrow:

\[
\begin{align*}
\triangleright^b \text{LftBorDead}(\kappa) & \Rightarrow \text{Nbor} \triangleright^b \text{LftBorDead}(\kappa) \ast \text{RawBor}(\kappa, P) \\
\triangleright \text{LftInh}(\kappa, P, \text{full}) & \Rightarrow \text{Ninh} \triangleright \text{LftInh}(\kappa, P, \text{full}) * ([\kappa] \Rightarrow \text{Nin} \triangleright P)
\end{align*}
\]

(4) just extends the \(B\) in 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:

\[
\begin{align*}
\triangleright \text{LftInh}(\kappa, P, \text{empty}) & \Rightarrow \text{Ninh} \triangleright \text{LftInh}(\kappa, P, \text{empty}) * \exists j. \text{OwnInh}(\kappa, o \{ j \}) * \text{BoxSlice}(\text{Ninh}, P, j) \\
\triangleright \text{LftBorAlive}(\kappa, P, B) & \Rightarrow \text{Nbor} \triangleright \text{LftBorAlive}(\kappa, P, B) * \&^\kappa_{\text{full}} P \\
\triangleright \text{LftVs}(\kappa, P, B) & \Rightarrow \text{LftVs}(\kappa, P, B) * \text{LftVs}(\kappa, P, P, B) \\
\end{align*}
\]
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 \( [\kappa] \Rightarrow \kappa \triangleright P \). From \( [\kappa] \) we learn that \( \kappa \) has ended, so we can get access to \( \text{LftDead}(\kappa) \). We can now apply the following lemma to the resources we got from (6):

\[
\triangleright \text{LftInh}(\kappa, P', \text{full}) \triangleright \text{OwnInh}(\kappa, \circ \{ j \}) \ast \text{BoxSlice}(\mathcal{N}_\text{inh}, P, j) \Rightarrow_{\mathcal{N}_\text{inh}} \triangleright P \ast \exists P'_j. \triangleright \text{LftInh}(\kappa, P'_j, \text{full})
\]

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

**Proof sketch of** \( \text{LftL-bor-fake} \). This is a trivial consequence of (4).

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

\[
\text{BoxSlice}(\mathcal{N}_\text{bor}, P, i) \vdash \triangleright \text{LftBorAlive}(\kappa, P_B) \ast \text{OwnBor}(\kappa, \circ i \leftarrow \text{in}, \text{inl}(\text{ex}()) \ast [\kappa]_q \Leftarrow_{\mathcal{N}_\text{bor}} \triangleright \text{LftBorAlive}(\kappa, P_B) \ast \text{OwnBor}(\kappa, \circ i \leftarrow \text{open}(q), \text{inl}(\text{ex}())) \ast \triangleright P
\]

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

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

To finish \( \text{LftL-bor-acc-cons} \), 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 \( \text{LftBorAlive} \). We use remove the empty slice from the borrow box to learn that \( P_B \) later decomposes into \( P^*_B = P \). Since \( \text{LftBorAlive} \) and \( \text{LftVs} \) are contractive in \( P_B \), we can rewrite with this decomposition. After adding a new empty slice to the box, we can obtain \( \triangleright \text{LftBorAlive}(\kappa_0, P^*_B \ast Q) \) (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

\[
\triangleright \text{LftVs}(\kappa, P'_{B} \ast Q) \ast \triangleright (\triangleright Q \ast [\kappa] \Rightarrow_{\mathcal{N}_\text{inh}} \triangleright P) \Rightarrow \triangleright \text{LftVs}(\kappa, P'_{B} \ast Q)
\]

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

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

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

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

For \( \text{LftL-bor-acc-atomic-cons} \), we instead continue like we did above with \( \text{LftL-bor-acc-cons} \): we remove the empty slice and create a new one with \( Q \) in it, and then we update \( \text{LftVs} \) 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 LftInv(κ'). If it is dead, then we use (4) to just "fake" &κ' _full P and &κ' _full Q, which can both be changed to borrows at κ using LftL-idx-shorten.

So we can assume we have \( \triangleright \text{LftAlive}(\kappa') \) and, in particular, \( \triangleright \text{LftBorAlive} \). We split the slice containing \( P \ast Q \) 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 &κ' _full P and &κ' _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 \( \kappa_0 \) and \( \kappa_1 \). We do have \( \kappa \subseteq \kappa_0 \) and \( \kappa \subseteq \kappa_1 \). We use reborrowing to obtain both borrows at lifetime \( \kappa' := \kappa \cap \kappa_0 \cap \kappa_1 \). Notice that we have \( \kappa \subseteq \kappa' \) Now we can start to do the actual merging. We check whether \( \kappa' \) is dead; if yes, so is \( \kappa \) and we can "fake" the result. So we obtain \( \triangleright \text{LftAlive}(\kappa') \) and, in particular, \( \triangleright \text{LftBorAlive} \). We merge the two slices containing \( P \) and \( Q \), respectively, and turn them into a slice containing \( P \ast Q \). We have all the fragments we need to fix up the authoritative \( B \), so we can close everything up again and obtain &κ' _full P \ast &κ' _full Q. 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 \( A \) in ALft and returning a singleton \( \kappa := \{A\} \).

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 \( \kappa \):

\[
(\forall \kappa', \kappa' \in \text{dom}(I) \land \kappa' \subseteq \kappa \Rightarrow \kappa' \in K) \land (\forall \kappa', \kappa' \in \text{dom}(I) \land \kappa \subseteq \kappa' \Rightarrow \kappa' \in K') \vdash \\
\text{LftAlive}(\kappa) \ast [\kappa] \Rightarrow \left( \begin{array}{c}
\bigotimes_{\kappa' \in K} \exists \kappa' \\text{LftAlive}(\kappa') \ast \bigotimes_{\kappa' \in K'} \text{LftDead}(\kappa')
\end{array} \right) \quad \text{LftDead}(\kappa)
\]

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

\[
(\forall \kappa \in K, \forall \kappa' \in \text{dom}(I), \kappa' \supseteq \kappa \Rightarrow \kappa' \in K) \land \\
(\forall \kappa \in K, \forall \kappa' \in \text{dom}(I), \text{LftAlive}(A, \kappa) \land \kappa' \notin K \land \kappa' \subseteq \kappa \Rightarrow \kappa' \in K') \vdash \\
\left( \begin{array}{c}
\bigotimes_{\kappa \in K} \text{LftAlive}(A, \kappa) \ast [\kappa] \Rightarrow \left( \begin{array}{c}
\bigotimes_{\kappa' \in K} \exists \kappa' \\text{LftAlive}(\kappa') \ast \bigotimes_{\kappa' \in K'} \text{LftDead}(\kappa')
\end{array} \right) \quad \text{LftDead}(\kappa)
\end{array} \right) 
\]

32
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 $\kappa$ such that $\text{LftAliveIn}(A, \kappa)$. If no, we have nothing to do. Otherwise, we select a minimal element $\kappa \in \{\kappa \in K \mid \text{Lft AliveIn}(A, \kappa)\}$ 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 $\kappa$ is not only minimal in $\{\kappa \in K \mid \text{Lft AliveIn}(A, \kappa)\}$, 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 $\kappa'$ will either also be in the down-closure of $K$ (and hence it is in $K'$), or it will be $\kappa$ 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 $\kappa$. To this end, we invoke (12). By our frame and by the fact that $\kappa$ is a minimal alive lifetime in $K$, we know that all lifetimes strictly shorter than $\kappa$ are in $K'$ and hence alive. Furthermore, we know that all lifetimes strictly larger than $\kappa$ are in $K''$ and hence dead. This completes the proof.

Now, we can come back to proving the closing view shift of $\text{LFTL-BEGIN}$. We start out assuming $[\{\Lambda\}]_1$. We open $\text{LftLCtx}$ and take a step to obtain $\text{LftLInv}$. From the token we own, we know $A(\Lambda) = \text{LftStAlive}$. We update our token to obtain $[\dagger\{\Lambda\}]$. Now we want to apply (13) with $K := \{\kappa \in I \mid \Lambda \in \kappa\}$ and $K' := \{\kappa \in \text{dom}(I) \mid \kappa \notin K \land \exists \kappa' \in K. \text{LftAliveIn}(A, \kappa') \land \kappa \subset \kappa'\}$. 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 \textit{\textbackslash{}lambda}_\text{Rust} \text{ 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 $\lambda_{\text{Rust}}$ 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 $\text{own} : \text{size} \in \mathbb{N}, \text{own} : \text{TId} \times \text{list}(\text{Val}) \rightarrow \text{iProp}, \text{shr} \in \text{Lft} \times \text{TId} \times \text{Loc} \rightarrow \text{iProp}$ such that

\[
\forall t, v. \text{size} (\text{Ty-size})
\]

\[
\forall \kappa, t, \ell. \text{shr} (\kappa, t, \ell) \Rightarrow \Box \text{shr} (\kappa, t, \ell) \quad (\text{TY-shr-persistent})
\]

\[
\forall \kappa, t, \ell. \text{full} (\ell \mapsto \text{own}(t)) * [\kappa]_q \Rightarrow \mathcal{N}_n \text{shr} (\kappa, t, \ell) * [\kappa]_q \quad (\text{TY-share})
\]

\[
\forall \kappa, \kappa', t, \ell. \kappa' \subseteq \kappa \Rightarrow \text{shr} (\kappa, t, \ell) \Rightarrow \text{shr} (\kappa', t, \ell) \quad (\text{TY-shr-mono})
\]

\text{Semantic Type} \quad \text{A semantic type} is a tuple $(\text{size} \in \mathbb{N}, \text{own} : \text{TId} \times \text{list}(\text{Val}) \rightarrow \text{iProp}, \text{shr} \in \text{Lft} \times \text{TId} \times \text{Loc} \rightarrow \text{iProp})$ such that

\[
\forall t, v. \text{size} (\text{Ty-size})
\]

\[
\forall \kappa, t, \ell. \text{shr} (\kappa, t, \ell) \Rightarrow \Box \text{shr} (\kappa, t, \ell) \quad (\text{TY-shr-persistent})
\]

\[
\forall \kappa, t, \ell. \text{full} (\ell \mapsto \text{own}(t)) * [\kappa]_q \Rightarrow \mathcal{N}_n \text{shr} (\kappa, t, \ell) * [\kappa]_q \quad (\text{TY-share})
\]

\[
\forall \kappa, \kappa', t, \ell. \kappa' \subseteq \kappa \Rightarrow \text{shr} (\kappa, t, \ell) \Rightarrow \text{shr} (\kappa', t, \ell) \quad (\text{TY-shr-mono})
\]

\text{Semantic type inclusion}

\[
\tau_1 \sqsubseteq^v \tau_2 := \tau_1.\text{size} = \tau_2.\text{size} \land (\Box \forall t, v. \tau_1.\text{own} (t, v) \Rightarrow \tau_2.\text{own} (t, v)) \land
\]

\[
(\Box \forall \kappa, t, \ell. \tau_1.\text{shr} (\kappa, t, \ell) \Rightarrow \tau_2.\text{shr} (\kappa, t, \ell))
\]

Figure 6: The semantic domain of types.
\[ \text{size}(\tau) = \lceil \text{size} \rceil. \]

### 5.1.1 Owned pointers and mutable references

With the foundations introduced above, we can now define

\[ \text{size}.own(\tau, v) := \exists \ell. v = \ell \rightarrow \lceil \text{size} \rceil \]

\[ \text{size}.\ell \rightarrow \tau.own(\tau, v) \]

Here and in the remainder of this document, \( \ell \rightarrow \Phi \) is sugar for \( \exists v. \ell \rightarrow v \rightarrow \Phi(v) \). 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 \&shr \( \tau \) is not given uniformly; instead, it is defined by the type itself:

\[ \lceil \&shr \tau \rceil \text{own}(\tau, v) := \exists \ell. v = \ell \rightarrow \lceil \text{shr} \rceil(\tau, \ell) \]

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).
\[
\llbracket \| \rrbracket_\gamma := \text{Type} \{ \text{size} := 0; \text{own} := \lambda_{\gamma, \cdot} \text{False}; \text{shr} := \lambda_{\gamma, \cdot} \text{False} \} \quad (\text{TY-DEF-EMP})
\]
\[
\llbracket \varnothing \rrbracket_\gamma := \text{Type} \{ \text{size} := 0; \text{own} := \lambda_{\gamma, \cdot} \varnothing, \varnothing = \emptyset; \text{shr} := \lambda_{\gamma, \cdot} \text{True} \} \quad (\text{TY-DEF-UNIT})
\]
\[
\llbracket \text{bool} \rrbracket_\gamma := \text{SimpleType}(\lambda_{\gamma, v. v = \text{true} \lor v = \text{false}}) \quad (\text{TY-DEF-BOOL})
\]
\[
\llbracket \text{int} \rrbracket_\gamma := \text{SimpleType}(\lambda_{\gamma, \exists z. v = z}) \quad (\text{TY-DEF-NAT})
\]
\[
\llbracket \text{own}_n \rrbracket_\gamma := \text{Type} \{ \text{size} := 1; \}
\]
\[
\text{own} := \lambda t, \varnothing. \exists \ell. \varnothing = [\ell] \star \ell \mapsto \llbracket \tau \rrbracket_\gamma.\text{own}(\ell) \ast \left(\llbracket \tau \rrbracket_\gamma.\text{size} = 0 \lor [n]_\gamma > 0 \ast \llbracket \tau \rrbracket_\gamma.\text{size}/[n]_\gamma, \ell \right)
\]
\[
\text{shr} := \lambda \kappa, t, \ell. \exists \ell' \cdot \& \frac{\lambda q', \ell \overset{q'}{\to} \ell'}{(\forall q'. [\kappa]_q \text{refl}_{\text{Nfin}, \text{Nfin}} \llbracket \tau \rrbracket_\gamma.\text{shr}(\kappa, t, \ell') \ast [\kappa]_q)} \quad (\text{TY-DEF-OWN})
\]
\[
\llbracket \& \text{mut} \rrbracket_\gamma := \text{Type} \{ \text{size} := 1; \}
\]
\[
\text{own} := \lambda t, \varnothing. \exists i. \varnothing = [i] \star \sum_{\ell=i} \llbracket \tau \rrbracket_\gamma.\text{own}(\ell, \varnothing); \quad \text{shr} := \lambda \kappa, t, \ell. \sum_{\ell<i} \llbracket \tau \rrbracket_\gamma.\text{shr}(\kappa, t, \ell + \sum_{j<i} \llbracket \tau \rrbracket_\gamma.\text{size}) \quad (\text{TY-DEF-SHR})
\]
\[
\llbracket \Sigma \rrbracket_\gamma := \text{Type} \{ \text{size} := 1 + \max \llbracket \tau \rrbracket_\gamma.\text{size}; \}
\]
\[
\text{own} := \lambda t, \varnothing. \exists \varnothing. \varnothing = [i] \overset{\sum_{\ell=i}}{\star} \llbracket \tau \rrbracket_\gamma.\text{own}(t, \varnothing) \ast [\varnothing'] = 1 + \max \llbracket \tau \rrbracket_\gamma.\text{size}; \quad \text{shr} := \lambda \kappa, t, \ell. \sum_{\ell<i} \llbracket \tau \rrbracket_\gamma.\text{shr}(\kappa, t, \ell + 1) \ast \& \frac{\lambda q, \ell \overset{q}{\to} i}{(\ell + 1 + \sum_{j<i} \llbracket \tau \rrbracket_\gamma.\text{size} \mapsto \lambda (\llbracket \tau \rrbracket_\gamma.\text{size} \mapsto \max \llbracket \tau \rrbracket_\gamma.\text{size})} \quad (\text{TY-DEF-PROD})
\]
\[
\llbracket \llbracket \llbracket \rrbracket \rrbracket_\gamma := \text{SimpleType}(\lambda_{\gamma, \cdot} \cdot \text{True}) \quad (\text{TY-DEF-UNUNIT})
\]
\[
\llbracket \llbracket \llbracket \rrbracket \rrbracket_\gamma := \Pi[\llbracket \llbracket \rrbracket \rrbracket_\gamma] \quad \text{of length } [\llbracket \llbracket \rrbracket \rrbracket_\gamma \text{length}] \quad (\text{TY-DEF-UNUNIT})
\]
\[
\llbracket \forall \mathbf{f} : E; \varnothing \mapsto \tau \rrbracket_\gamma := \text{SimpleType}(\lambda_{\gamma, v. \exists f, \varnothing, k, F, v = \text{funrec } f(\varnothing) \text{ ret } k := F \ast \lambda f, \kappa, v_k, \varnothing. (\forall f : \llbracket \llbracket \rrbracket \rrbracket_\gamma \mid k < \text{cont}(f \llbracket \llbracket \rrbracket_\gamma) ; \exists x < \text{own } \varnothing) ; \varnothing \llbracket \llbracket \rrbracket \rrbracket_\gamma \mapsto \text{funrec } f(\varnothing) \text{ ret } k := F/f, \varnothing/\varnothing, v_k/k]) \quad (\text{TY-DEF-FN})
\]
\[
\llbracket \mu T. \tau \rrbracket_\gamma := \text{fix}(\lambda T. \llbracket \tau \rrbracket_\gamma \mid T \to \tau) \quad 36
\]

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 \( \text{int} \). We define ownership and sharing of a number as follows:

\[
\begin{align*}
\text{int}.\text{own}(t, \overline{v}) & := \exists z. \overline{v} = [z] \\
\text{int}.\text{shr}(\kappa, _t, \ell) & := \exists \overline{v}. \& \frac{\text{frac}}{(\lambda q. \ell [q \mapsto \overline{v})} * [\text{int}].\text{own}(t, \overline{v})
\end{align*}
\]

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

\[
\text{SimpleType}(\Phi) := \text{Type}\{\text{size} := 1; \text{own} := \lambda t, \overline{v}. \exists v. [v] * \Phi(t, v); \text{shr} := \lambda \kappa, t, \ell. \exists \overline{v}. \& \frac{\text{frac}}{(\lambda q. \ell [q \mapsto \overline{v})} * \Phi(t, v)\}
\]

For a simple type, we only have to show that the ownership predicate \( \Phi \) is persistent and that \( \text{Ty-size} \) 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:

\[
\begin{align*}
\text{[own]\ } \kappa.\text{shr}(\kappa, t, \ell) & := \exists \ell'. \& \frac{\text{frac}}{(\lambda q. \ell [q \mapsto \ell')} * (\forall q. [\kappa]_q \Rightarrow \begin{cases} \text{N}_\text{shr}.\text{N}_\kappa \kappa \text{ \kappa.shr} \kappa.\text{shr}(\kappa, t, \ell') & \text{if } [\kappa]_q \end{cases}) \\
\text{[mut]\ } \kappa.\text{shr}(\kappa', t, \ell) & := \exists \ell'. \& \frac{\text{frac}}{(\lambda q. \ell [q \mapsto \ell')} * (\forall q. [\kappa \cap \kappa']_q \Rightarrow \begin{cases} \text{N}_\text{shr}.\text{N}_\kappa \kappa \text{ \kappa.shr} \kappa.\text{shr}(\kappa \cap \kappa', t, \ell') & \text{if } [\kappa \cap \kappa']_q \end{cases})
\end{align*}
\]

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 \([\ell].\text{shr}(\kappa, t, \ell')\). Unfortunately, that would make it impossible to prove \(\text{Ty-share}\). Since owned pointers are, well, pointers, they only own the data they point to \(\text{later} or under a borrow\), 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 across 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.

\[
\begin{align*}
\llbracket \Pi \rrbracket \text{.own}(t, \overline{\pi}) & := \exists \overline{\pi}. \overline{\pi} = \sum_i \overline{\pi}_i \ast \llbracket \tau_i \rrbracket \text{.own}(t, \overline{\pi}_i) \\
\llbracket \Pi \rrbracket \text{.shr}(\kappa, t, \ell) & := \bigotimes_i \llbracket \tau_i \rrbracket \text{.shr}(\kappa, t, \ell + \sum_{j < i} \llbracket \tau_j \rrbracket \text{.size}) \\
\llbracket \Sigma \rrbracket \text{.own}(t, \overline{\pi}) & := \exists \overline{\pi}, \overline{\pi}' . \overline{\pi} = [i] \bowtie \overline{\pi}' \bowtie \overline{\pi}'' \ast |\overline{\pi}''| = 1 + \max_j \llbracket \tau_j \rrbracket \text{.size} \ast \llbracket \tau_i \rrbracket \text{.own}(t, \overline{\pi}') \\
\llbracket \Sigma \rrbracket \text{.shr}(\kappa, t, \ell) & := \exists i . \& \text{trac} (\lambda q . \ell \xrightarrow{q} i \ast (\ell + 1 + \llbracket \tau_i \rrbracket \text{.size} \mapsto \lambda \overline{\pi} . \llbracket \tau_i \rrbracket \text{.size} + |\overline{\pi}|) = \max_j \llbracket \tau_j \rrbracket \text{.size}) \ast \llbracket \tau_i \rrbracket \text{.shr}(\kappa, t, \ell + 1)
\end{align*}
\]

5.1.4 Copy, Send, Sync

\[
\begin{align*}
\llbracket \tau \text{ copy} \rrbracket_\gamma & := (\Box \forall t. \llbracket \tau \rrbracket_\gamma \text{.own}(t, \overline{\pi}) \Rightarrow \Box \llbracket \tau \rrbracket_\gamma \text{.own}(t, \overline{\pi})) \land \\
\Box \forall \kappa, t, \ell, q . \llbracket \tau \rrbracket_\gamma \text{.shr}(\kappa, t, \ell) & \Rightarrow (\Box \text{Na} : t . \text{Nshr} \llbracket \gamma \rrbracket \geq \ell, \llbracket \tau \rrbracket_\gamma \text{.size}) \ast [\kappa]_q \Leftrightarrow \\
q' . \text{Na} : t . \text{Nshr} \llbracket \gamma \rrbracket (\ell + \llbracket \tau \rrbracket_\gamma \text{.size}) \ast \ell \xrightarrow{q'} \llbracket \tau \rrbracket_\gamma \text{.own}(t)) \land \text{Nshr} \land \text{Ninst} \\
\llbracket \tau \text{ send} \rrbracket_\gamma & := \Box \forall t_1, t_2, \overline{\pi} . \llbracket \tau \rrbracket_\gamma \text{.own}(t_1, \overline{\pi}) \Rightarrow \llbracket \tau \rrbracket_\gamma \text{.own}(t_2, \overline{\pi}) \\
\llbracket \tau \text{ sync} \rrbracket_\gamma & := \Box \forall \kappa, t_1, t_2, \ell . \llbracket \tau \rrbracket_\gamma \text{.shr}(\kappa, t_1, \ell) \Rightarrow \llbracket \tau \rrbracket_\gamma \text{.shr}(\kappa, t_2, \ell)
\end{align*}
\]

5.2 Type and continuation contexts

\[
\begin{align*}
\llbracket \mathbf{T} \rrbracket_\gamma : TId & \rightarrow iProp \\
\llbracket \mathbf{0} \rrbracket_\gamma (t) & := \text{True} \\
\llbracket \mathbf{T}, p < \tau \rrbracket_\gamma (t) & := \llbracket \tau \rrbracket_\gamma \text{.own}(t, \llbracket p \rrbracket) \ast \llbracket \mathbf{T} \rrbracket_\gamma (t) \\
\llbracket \mathbf{T}, p \llbracket \llbracket \llbracket \llbracket \mathbf{L} \rrbracket_\gamma (1) \ast \llbracket \mathbf{T} \rrbracket_\gamma \langle \pi \rangle (t) \Rightarrow \text{wp} \llbracket k \rrbracket_\gamma (\pi) \{ \text{True} \} \rangle \ast \llbracket \mathbf{T} \rrbracket_\gamma (t)
\end{align*}
\]

\[
\begin{align*}
\llbracket \mathbf{K} \rrbracket_\gamma : TId & \rightarrow iProp \\
\llbracket \mathbf{0} \rrbracket_\gamma (t) & := \text{True} \\
\llbracket \mathbf{K}, k < \text{cont}(\mathbf{L}, \pi, \mathbf{T}) \rrbracket_\gamma (t) & := (\forall \tau . \llbracket \text{Na} : t \ast \llbracket \mathbf{L} \rrbracket_\gamma (1) \ast \llbracket \mathbf{T} \rrbracket_\gamma \langle \pi \rangle (t) \Rightarrow \text{wp} \llbracket k \rrbracket_\gamma (\pi) \{ \text{True} \}) \ast \llbracket \mathbf{K} \rrbracket_\gamma (t)
\end{align*}
\]

38
5.3 Lifetime contexts and judgments

The local and external lifetime contexts are interpreted as follows:

\[ [E]_\gamma : iProp \]
\[ [\emptyset]_\gamma := \text{True} \]
\[ [E, \kappa \sqsubseteq \kappa']_\gamma := [\kappa]_\gamma \sqsubseteq [\kappa']_\gamma \ast [E]_\gamma \]

\[ [L]_\gamma : Fract \rightarrow iProp \]
\[ [\emptyset]_\gamma (q) := \text{True} \]
\[ [L, \kappa \sqsubseteq \kappa']_\gamma (q) := \exists \kappa'. [\kappa]_\gamma = \kappa' \cap (\cap \tau_\gamma) \ast [\kappa']_\gamma \ast \Box (\Box [\kappa']_\gamma) \ast [L]_\gamma (q) \]

\[ [E_1; L_1 \vdash E_2] := \forall q. [L]_\gamma (q) \rightarrow \Box ([E_1]_\gamma \rightarrow [E_2]_\gamma) \]
\[ [E; L \vdash \kappa_1 \sqsubseteq \kappa_2] := \forall q. [L]_\gamma (q) \rightarrow \Box ([E]_\gamma \rightarrow [\kappa_1]_\gamma \sqsubseteq [\kappa_2]_\gamma) \]
\[ [E; L \vdash \kappa \text{ alive}] := \forall q. [E]_\gamma \rightarrow ([L]_\gamma (q) \iff q'. [\kappa]_\gamma (q)) \]

5.4 Judgments

\[ [E; L \vdash \tau_1 \Rightarrow \tau_2]_\gamma := \forall q. [L]_\gamma (q) \rightarrow \Box ([\tau_1]_\gamma \rightarrow \tau_2) \]
\[ [E; L \vdash T_1 \Rightarrow T_2]_\gamma := \forall t, q. [E]_\gamma \ast [L]_\gamma (q) * [T_1]_\gamma (t) \Rightarrow [T_2]_\gamma (t) \]
\[ [T_1 \Rightarrow^\kappa T_2]_\gamma := \forall t. [\tau_1]_\gamma \ast [T_1]_\gamma (t) \Rightarrow [T_2]_\gamma (t) \]
\[ [E; K_1 \vdash K_2]_\gamma := \forall t. [\kappa_1]_\gamma \ast [K_1]_\gamma (t) \Rightarrow [K_2]_\gamma (t) \]
\[ [E; L \vdash \tau_1 \rightarrow^\tau \tau_2]_\gamma := \forall t, q. [E]_\gamma \ast [L]_\gamma (q) \ast [\tau_1]_\gamma \ast \text{own}(t, [\nu]) \Rightarrow [\tau_2]_\gamma \ast \text{own}(t, [\nu]) \]
\[ \exists \nu, \ell, q'. \ell = v \ast \nu \ast \nu \ast [\tau]_\gamma \ast \text{size} \ast \ell \rightarrow \nu \ast (\nu \ast [\nu]_\gamma \ast \text{own}(t, \nu)) \]
\[ [E; L \vdash \tau_1 \lnot \tau_2]_\gamma := \forall t, q. [E]_\gamma \ast [L]_\gamma (q) \ast [\nu : t]_\gamma \ast \text{own}(t, [\nu]) \Rightarrow [\tau_2]_\gamma \ast \text{own}(t, [\nu]) \]
\[ \exists \nu, \ell, q'. \ell = v \ast \nu \ast \nu \ast [\tau]_\gamma \ast \text{size} \ast \ell \rightarrow \nu \ast (\nu \ast [\nu]_\gamma \ast \text{own}(t, \nu)) \]

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 \([\_\_]_\gamma\), the resulting Iris theorem holds.

Theorem 2 (Adequacy of the logical relation). Let \(f\) be a \(\lambda_{\text{Rust}}\) function such that the Iris assertion \([\emptyset; \emptyset \vdash f \vdash x. x \sim \text{fn}(\t) \rightarrow \text{Pi}]\) holds, then when we execute \(f(\lambda x. x)\) (passing it the default continuation), no execution ends in a stuck state.