Flux: Liquid Types for Rust

NICO LEHMANN, UC San Diego, USA
ADAM GELLER, University of British Columbia, Canada
GILLES BARTHE, Max-Planck Institute, Germany
NIKI VAZOU, IMDEA, Madrid, Spain
RANJIT JHALA, UC San Diego, USA

Low-level pointer-manipulating programs are hard to verify, requiring complex spatial program logics that support reasoning about aliasing and separation. Worse, when working over collections, these logics burden the programmer with annotating loops with quantified invariants that describe the contents of the collection. We present Flux, which shows how logical refinements can work hand in glove with Rust's ownership mechanisms to yield ergonomic type-based verification for low-level imperative code. To do so Flux *indexes* mutable locations, with pure (immutable) values that can appear in the refinements, and then exploits Rust's ownership mechanisms to abstract sub-structural reasoning about locations within Rust's polymorphic type constructors, extended by a notion of strong references to idiomatically track strong updates. By factoring complex invariants into type constructors and simple refinements, Flux can efficiently synthesize loop annotations via liquid inference. We use a suite of vector-manipulating programs to demonstrate the advantages of Flux's refinement types over program logics, as implemented in the state-of-the-art Prusti verifier. We show that liquid typing makes verification ergonomic by slashing *specification* lines by a factor of two, *verification* time by an order of magnitude, and *invariant annotation* overhead from 14% of code size, to nothing at all.

flux (/flʌks/) *n.* **1** a flowing or flow. **2** a substance used to refine metals. *v.* **3** to melt; make fluid.

1 INTRODUCTION

Low-level, pointer-manipulating programs are tricky to write and devilishly hard to verify, requiring complex *spatial* program logics that support reasoning about aliasing [O'hearn 2004; Reynolds 2002]. The Rust programming language [Matsakis and Klock II 2014] uses the mechanisms of *ownership types* [Clarke et al. 1998; Noble et al. 1998] to abstract fast pointer-based libraries inside typed APIs that let clients write efficient applications with static memory and thread safety. Recent systems like Prusti [Astrauskas et al. 2019], RustHorn [Matsushita et al. 2021], and Creusot [Denis 2020] have taken advantage of these ownership mechanisms to shield the programmer from some spatial assertions helping them instead focus on writing pure, first-order logic specifications which can be automatically verified by a solver.

Even with these advances, verification remains unpleasant. The programmer is still encumbered with providing verbose *annotations* to persuade the solver of the legitimacy of their code. For instance, when working over collections, program-logic based methods require the use of *loop invariants* that are *universally quantified* to account for the potentially unbounded contents of the collection. Such invariants often require a sophisticated understanding of the underlying spatial program logic, and worse, the quantification makes them difficult to synthesize.

Liquid refinements have obviated these problems in the purely *functional* setting [Constable and Smith 1987; Rushby et al. 1998; Xi and Pfenning 1999a]. Refinements express complex invariants by *composing* type constructors with simple quantifier-free logical predicates. Thus, they let us use syntax

Authors' addresses: Nico Lehmann, UC San Diego, USA, nlehmann@eng.ucsd.edu; Adam Geller, University of British Columbia, Canada, atgeller@cs.ubc.ca; Gilles Barthe, Max-Planck Institute, Germany, gilles.barthe@mpi-sp.org; Niki Vazou, IMDEA, Madrid, Spain, niki.vazou@imdea.org; Ranjit Jhala, UC San Diego, USA, rjhala@eng.ucsd.edu.

directed subtyping to *decompose* complex reasoning about those invariants into efficiently decidable (quantifier free) validity queries over the predicates, thereby enabling Horn-clause based annotation synthesis which makes verification ergonomic [Rondon et al. 2008]. Sadly, refinements have remained a fish out of water in the *imperative* setting. Mutation *changes* the type of variables, and aliasing makes it difficult to *track* those changes, making it hard for types to soundly *depend* on the shifting sands of program values. Systems like [Bakst and Jhala 2016; Rondon et al. 2010; Sammler et al. 2021; Toman et al. 2020] attempted to bridge the gap between pure refinements and impure heap locations using sub-structural type systems, but proved impractical as the retrofitted effect systems complicate specifications with non-idiomatic spatial constraints.

In this paper, we introduce Flux, which shows how refinements can work hand in glove with ownership mechanisms to yield ergonomic type-based verification for imperative (safe) Rust. Via three concrete contributions, we show how Flux lets the programmer abstract fast low-level libraries in *refined* APIs so that static typing yields application level correctness guarantees with minimal programmer annotation overhead.

1. Design (§ 3) Our first contribution is a type system design that seamlessly extends Rust's types with refinements in three steps. Following previous systems [Bakst and Jhala 2016; Sammler et al. 2021], Flux starts by *indexing* mutable locations, with pure (immutable) values that can appear in the refinements. Next, crucially, Flux shows how to exploit Rust's ownership mechanisms to encapsulate locations, thereby abstracting sub-structural reasoning within inside Rust's *type constructors*. Finally, Flux extends and refines Rust's mutable references with a notion of *strong references* that precisely and automatically track strong updates that alter the type of the mutated object.

2. Implementation (§ 4) Our second contribution is an implementation of the declarative type system as a plug-in to the Rust compiler. Flux works in three phases. In the first *spatial* phase, Flux automatically uses the function signatures to infer a mapping between program identifiers and heap locations, and the precise points where the refinements on a location may be assumed and must be asserted. At this juncture, the intermediate refinements are still unknown. Thus, in the second *checking* phase we perform refinement type checking using Horn variables for the unknown refinements, generating a system of Horn constraints, a solution to which implies the program is well-typed. Finally, in the third *inference* phase, we use a fixpoint computation to solve the constraints to verify the program or pinpoint an error when no solution exists. Crucially, factoring complex invariants into type constructors and simple refinements lets the solver efficiently synthesize solutions from a small set of quantifier-free templates [Cosman and Jhala 2017].

3. Evaluation (§ 5) Our third contribution is an empirical evaluation that demonstrates the advantages of Flux's refinement type-based verification over program logic based approaches. To do so, we use Flux and Prusti [Astrauskas et al. 2019], a state-of-the-art Rust verifier, to prove the absence of index-overflow errors in a suite of vector-manipulating programs. Our evaluation qualitatively illustrates how Flux's type-based APIs, naturally express invariants of containers and which (unbounded) areas of a container remains unchanged, both of which must be spelled out via complex quantified contracts in Prusti. More importantly, we show how liquid typing makes verification ergonomic by slashing verification time by an order of magnitude, specification sizes by a factor of 2, and shrinking the loop-invariant annotation overhead from nearly 14% of code size, to nothing at all.

```
#[lr::sig(fn(i32<@n>) -> bool<0 < n>)]
fn is_pos(n: i32) -> bool {
  if 0 < n { true } else { false }
}
```

```
#[lr::sig(fn(i32<@x>) -> i32{v:x<=v && 0<=v})]
fn abs(x: i32) -> i32 {
  if x < 0 { -x } else { x }
}
```

Fig. 1. Examples illustrating the three fundamental refinement type extensions in Flux: indexed types, refinement parameters and existential types.

2 A TOUR OF FLUX

Let us begin with a high-level overview of Flux's key features that illustrates how liquid refinements work hand in glove with Rust's types to yield a compact way to *specify* correctness requirements and an automatic way to *verify* them with minimal programmer overhead. First, we show how Flux decorates types with logical *refinements* that capture invariants (§ 2.1). Next, we demonstrate how Rust's type system allows us to precisely track refinements in the presence of imperative mutation and *updates* (§ 2.2). Finally, we show how Flux lets the programmer compose types to write refined *interfaces* that can then be used to verify clients in a modular fashion (§ 2.3).

2.1 Refinements

Refinement types allow expressions in some underlying, typically decidable, logic to be used to *constrain* the set of values inhabited by a type, thereby tracking additional information about the values of the type they refine [Jhala and Vazou 2021].

Indexed Types An indexed type [Xi and Pfenning 1999a] in Flux refines a Rust base type by indexing it with a refinement value. Each indexed type is associated with a refinement *sort* and it must be indexed by values of that sort. The exact meaning of the index varies depending on the type. For example, Rust primitive integers can be indexed by integers in the logic (of sort **int**) describing the exact integer they are equal to. Hence, indexed integers correspond to singleton types, for instance, the type `i32<n>` describes 32-bit[1] signed integers equal to `n` and the type `usize<n+1>` describes a pointer-sized unsigned integer equal to `n + 1`. Consequently, Flux can verify that the Rust expression $1 + 2 + 3$ has the type `i32<6>`. Similarly, the boolean type `bool<b>` is indexed by the boolean value `b` (of sort **bool**) it is equal to. For example, Flux can type the Rust expression `1 + 2 + 3 <= 10` as `bool<true>`. As a final example, we also consider the type `RVec<T, n>` of *growable* vectors indexed by their length (of sort **int**), as detailed later in § 2.3. For simplicity, our examples have a single index. However, as discussed in § 2.3, types can have multiple indices, for example, we can index a type for 2-D matrices by the number of rows and columns.

Refinement Parameters Flux's function signatures can be parameterized by variables in the refinement logic. Informally, such *refinement parameters* behave like ghost variables that exist solely for verification, but do not exist at runtime. Flux automatically instantiates the refinement parameters using the actual arguments passed in at the respective sites.

The first two features—indices and refinement parameters—are illustrated by the signature for the function `is_pos` specified with the Rust attribute `#[lr::sig(...)]` on the left in fig. 1. The function `is_pos` tests whether a 32-bit signed integer is (strictly) positive. The signature uses a refinement parameter `n` to specify that the function takes as input an integer equal to `n` and returns a boolean equal to `0 < n`. The syntax `@n` is used to bind and quantify over `n` for the entire scope of the function.

[1]Rust has primitive types for signed and unsigned integers of 8, 16, 32, 64 and 128 bits, plus the pointer-sized integers `usize` and `isize`.

Existential Types Indexed types suffice when we know the exact value of the underlying term, *i.e.,* can represent it with a *singleton* expression in the refinement logic. However, often we want to specify that the underlying value is from a *set* denoted by a refinement constraint [Constable and Smith 1987; Rushby et al. 1998]. Flux accomodates such specifications via *existential types* of the form `{v. B<v> | p}` where: (1) `v` is a variable in the refinement logic, (2) `B<v>` is a base type indexed by `v` and (3) `p` a predicate constraining `v`. For example, we can use the existential type `{v. i32<v> | 0 < v}` to specify the set of *positive* 32-bit integers. Similarly, we can describe the set of *non-empty* vectors with the type `{v. RVec<T, v> | 0 < v}`. For types with only one index, we define the syntax `B{v : p}` to mean `{v. B<v> | p}`. Hence, we can write the two types above as `i32{v : v > 0}` and `RVec<T>{v : 0 < v}`. Further, we write `B` to abbreviate `B{v : true}`.

Existential types are illustrated by the signature for the function `abs` shown on the right in fig. 1 which computes the absolute value of the `i32` input `x`. The function's output type is an existential that specifies that the returned value is a non-negative `i32` whose values is at least as much that of the input `x`.

2.2 Updates

The whole point of Rust, of course, is to allow for efficient imperative *sharing* and *updates*, without sacrificing thread- or memory-safety. This is achieved via an *ownership type system* that ensures that aliasing and mutation cannot happen at the same time [Clarke et al. 2013; Jung et al. 2017; Noble et al. 1998]. Next, let's see how Flux lets logical constraints ride shotgun with Rust's ownership types to scale refinement types to an imperative setting using (parts of) an implementation of the k-means clustering algorithm shown in fig. 2. In this code, each n-dimensional point is represented as a vector of size n, and we wish to verify the safety of vector accesses, *i.e.,* that each read and write occurs within the bounds of the vector.

This goal is challenging for two reasons. First, the vector sizes can *grow* or *shrink* via `push` or `pop` operations. Second, the code maintains several *collections* of points (*e.g.,* clusters or candidate centers) where each point must have the same dimensions. This combination of mutation and collections complicates verification using program-logic [Astrauskas et al. 2019] by necessitating the use of complex, user-specified, universally quantified invariants (§ 5). In contrast, let us see how Flux entirely automates verification by fusing refinements with Rust's syntactic discipline for controlling sharing and updates.

Exclusive Ownership Rust's most basic form of ownership is *exclusive ownership*, in which only one function has the right to mutate a memory location. In Flux, exclusive ownership plays crucial role: by ruling out aliasing, we can safely perform *strong updates* [Ahmed et al. 2007; Smith et al. 2000], *i.e.,* we can change the refinements on a type when updating data, and thereby, use different types to denote the different values at that location at different points in time.

For example, consider the function `init_zeros` which takes as input a `usize` value `n` and returns as output an `n`-dimensional vector of 32-bit floats (`f32`), specified as `RVec<f32, n>`. The integer counter `i` is initialized with `0` on line 4 of fig. 2. After the definition, the memory location pointed to by `i` is exclusively owned by the function `init_zeros`, meaning it cannot be updated by any other code or thread. Thus, if at line 7 in fig. 2 we know `i` has type `i32<j>` for some `j`, then, after executing the statement `i += 1`, we can update its type to be `i32<j + 1>`. As detailed later in § 4 Flux's liquid typing exploits these strong updates to automatically infer that `i` is equal to the length of `vec` (and since the loop exists with `i = n`, the returned value `vec` has type `RVec<f32, n>`).

Shared and Mutable References Exclusive ownership suffices for local updates (*e.g.,* `i` in `init_zeros`) but for more complex data, ultimately, functions must reliquish ownership to other functions that update the data in some fashion. Rust's unique approach to allow this is called *borrowing*, via two

```rust
#[lr::sig(fn(usize<@n>) -> RVec<f32, n>)]
fn init_zeros(n: usize) -> RVec<f32> {
    let mut vec = RVec::new();
    let mut i = 0;
    while i < n {
        vec.push(0.0);
        i += 1;
    }
    vec
}

#[lr::sig(fn(&mut RVec<f32, @n>, &RVec<f32, n>))]
fn add(x: &mut RVec<f32>, y: &RVec<f32>) {
    let mut i = 0;
    while i < x.len() {
        *x.get_mut(i) = *x.get(i) + *y.get(i);
        i += 1;
    }
}

#[lr::sig(fn(usize<@n>, &mut RVec<RVec<f32, n>, @k>, &RVec<f32, k>))]
fn normalize_centers(n: usize, xs: &mut RVec<RVec<f32>>, ws: &RVec<usize>) -> i32 {
    let mut i = 0;
    while i < xs.len() {
        normal(xs.get_mut(i), *ws.get(i));
        i += 1;
    }
}
```

Fig. 2. Code taken from an implementation of the k-means clustering algorithm illustrating the interaction between ownership and refinement types. The code uses `RVec`, a refined version of Rust's `Vec`, to ensure the algorithm works over vectors of the same dimension.

kinds of references. First, a value of type `&T` is a *shared* reference, that can be used to access the `T` value in a read-only fashion. Second, a value of type `&mut T` is considered to be a *mutable reference* that can be used to write or update the contents of a `T` value. For safety, Rust allows multiple aliasing (read-only) shared references but only a single mutable reference to a value at a time.

For example, consider the function `add` in fig. 2, which adds two vectors point-wise, mutating the first vector in-place. This means that clients of `add` must grant access to mutate the vector. Rust's key idea is that when calling `add` the vector is only temporarily borrowed, granting write access to the callee for the duration of the function call. In plain Rust, this protocol is specified by saying that `add` takes a mutable reference `&mut RVec<f32>` to the first vector and a shared reference `&RVec<f32>` to the second (which is not modified by `add`). Flux refines the Rust signature to say that the first argument has type `&mut RVec<f32, @n>`, *i.e.,* is a vector of size `n`, and that the second vector also has the same size. Crucially, unlike updates to an owned location, updates through a mutable reference `&mut T` *do not change* the type `T`. In other words, mutating through a mutable reference can only perform *weak updates*. This may sound like a restriction, but it is actually a

```
impl RVec<T, @n> {
    fn new() -> RVec<T, 0>;
    fn len(&self) -> usize<n>;
    fn is_empty(&self) -> bool<n == 0>;
    fn get(&self, idx: usize{v : v < n}) -> &T;
    fn get_mut(&mut self, idx: usize{v : v < n}) -> &mut T;
    fn push(&strg self, value: T) -> ()
       ensures *self: RVec<T, n + 1>;
    fn pop(&strg self) -> T
       requires n > 0 ensures *self: RVec<T, n - 1>;
}
```

Fig. 3. Signatures of `RVec` methods. The `impl` syntax mirrors `impl` blocks in Rust. The type variable `T` and the refinement variable `n` are in scope for the entire block. The syntax `&self`, `&mut self`, and `&strg self` are used as in Rust as shorthand for reference of the block's type (`RVec<T, n>` in this case).

powerful notion, as the caller can rely upon the fact that recipients of a mutable reference `&mut T` will respect the invariants of `T`. For instance, clients of `add` can be sure that the first vector will continue to have the same length after the function returns.

Whither Strong Updates? Rust's existing mechanisms to control aliasing and mutation—(1) exclusive ownership, which provide strong updates but preclude any aliasing, (2) shared references (borrows), which allow aliasing, but preclude any form of updates, and (3) mutable references (borrows), which allow weak updates, but preclude any form of aliasing—are insufficient for refinement typing. To verify idiomatic Rust code we require references that permit *strong updates*.

Consider again the function `init_zeros` from fig. 2. Initially, `vec` is exclusively owned by the function after its definition in line 3. Later, in line 6, we call the `push` method on `vec`. The syntax `vec.push(0.0)` desugars to `RVec::push(&mut vec, 0.0)`, where `&mut vec` creates a mutable reference to `vec`. Thus, the traditional way to handle this call would be for `push` to take a mutable reference. However, as mutable references only allow weak updates, we would not be able to change the type of `vec` after the call, thereby leaving the verifier clueless that the size increased by one.

Strong References To this end, Flux extends Rust with *strong references*, written `&strg T`, which refine Rust's `&mut T` and, like regular mutable references, also grant temporary exclusive access but allow strong updates. Flux accommodates strong references by extending function signatures to specify the *updated type* of each strong references after the function returns.

For example, consider the signature for `RVec::push` shown in fig. 3. The plain Rust signature for this function is `fn(&mut self, T) -> ()`, where `self` refers to the receiver `RVec<T>`. However, the Flux signature for `RVec::push` refines the above to specify that: (1) the receiver is an `RVec<T, n>`, *i.e.,* a vector of size `n`, (2) `self` is a *strong* mutable reference, and (3) the *updated* type of the receiver is `RVec<T, n+1>`, as denoted by the method's `ensures` clause. Crucially, as `n` is a refinement parameter denoting the input size of the receiver, the above updated type precisely tracks how `push` increases the size of the underlying vector.

2.3 Interfaces

Our design for Flux crucially allows us to use standard type constructors as building blocks to *compose* complex structures from simple ones, in a way that, dually, lets standard syntax directed typing rules *decompose* complex reasoning about those structures into efficiently decidable (quantifier free) validity queries over the constituents. Next, we illustrate this composition by showing

a refined API for operating on *vectors*, and demonstrating how the API can be used to implement a refined API for reasoning about *matrices*, all while guaranteeing that well-typed programs access the underlying containers within their prescribed bounds.

A Refined Vector API Figure 3 summarizes the signatures for `RVec`—vectors refined by size. Flux uses the syntax `impl RVec<T, @n>`, inspired by Rust `impl` blocks, to bind the type variable `T` and the refinement variable `n` for the entire block. In the signature for each method, the syntax `&self`, `&mut self`, and `&strg self` denote references to the receiver's type, `RVec<T, n>` in this case.

- `new` is used to construct empty vectors: the return type `RVec<T, 0>` guarantees the returned vector has size zero.
- `len` can be used to determine the size of the vector. The method takes a shared reference, which implicitly specifies the vector will have the same length after the function returns. Moreover, the returned `usize<n>` stipulates the result equals the receiver's size.
- `is_empty` also takes a shared reference and checks if the vector is empty, returning a boolean which is true exactly when the length is zero.
- `get` and `get_mut` are used to access the elements of the vector: `get` returns a shared (read-only) reference while `get_mut` returns a mutable one that can be used to update the vector (as in line 16 in `add` of fig. 2). The type for the index `idx` specifies that only valid indices (less than size `n`) can be used to access the receiving vector. Crucially, by taking a mutable reference, `get_mut` guarantees that the length of the vector and the type of the elements it contains remain the same after the function returns. Furthermore, by returning a mutable reference, users of `get_mut` must respect the invariants in `T` when mutating the reference, ensuring the vector will continue to hold elements of type `T`.
- `push` and `pop` are used to grow and shrink the vector by one element at a time. As discussed in § 2.2, `push` takes a strong reference and specifies that the length of the vector has increased by one after the function returns, via the `ensures` clause `*self: RVec<T, n + 1>` which denotes the updated type of the location pointed to by the `self` parameter after the function call. Similarly, `pop` ensures the length has decreased by one, but stipulates the precondition that the vectors must be non-empty before popping via the `requires` clause that constrains the refinement parameter `n` to be positive.

Quantified Invariants via Polymorphism `RVec` is polymorphic over `T`: the type of elements it contains. For a classical type systems, polymorphism facilitates code *reuse*: we can use the datatype to hold collections of integers or strings or booleans *etc.*. Flux exploits polymorphism to enable specification and verification: we can instantiate `T` with arbitrary refined types to compactly specify that *all* elements of the vector satisfy some invariant. For instance, the function `normalize_centers` in fig. 2 uses `RVec<RVec<f32, n>, k>` to concisely specify a collection of `k`-centers, *each of which* is an `n`-dimensional point. Program logic based methods must use universally quantified formulas to express such properties, which increases the specification burden on programmers (who must now write tricky quantified invariants), and the verification burden on the solver (which must now reason about those quantified invariants!). In contrast, Flux's type-directed method automatically verifies, that despite working over mutable references, we can be sure all the inner vectors still have the same length after the function returns even though we are passing references to these vectors to the function `normal` in line 25.

A Refined Matrix API Suppose now that we want to work with 2-dimensional matrices. As we have seen in the `kmeans` code we could implement this as a vector of vectors, but spelling out the type every time gets tedious. By extending Rust's built-in mechanism for custom data structure with refinement types, Flux lets us easily define refined abstractions that avoid working directly over the vector of vectors. The code in fig. 4 uses `RVec` to build a verified refined API for matrices.

```
#[lr::refined_by(m: int, n: int)]
pub struct RMat {
    #[lr::field(RVec<RVec<f32, n>, m>)]
    vec: RVec<RVec<f32>>
}

impl RMat {
    #[lr::sig(usize<@m>, usize<@n>) -> RMat<m, n>]
    pub fn new(m: usize, n: usize) -> RMat {
        let mut vec = RVec::new();
        let mut i = 0;
        while i < m {
            vec.push(init_zeros(n));
            i += 1;
        }
        RMat { vec }
    }

    #[lr::sig(&RMat<@m, @n>, usize{v : v < m}, usize{v : v < n}) -> &f32]
    pub fn get(&self, i: usize, j: usize) -> &f32 {
        self.vec.get(i).get(j)
    }

    #[lr::sig(&mut RMat<@m, @n>, usize{v : v < m}, usize{v : v < n}) -> &mut f32]
    pub fn get_mut(&mut self, i: usize, j: usize) -> &mut f32 {
        self.vec.get_mut(i).get_mut(j)
    }
}
```

Fig. 4. Implementation of a 2-dimensional matrix as a vector of vectors of the same length. The attribute `lr::refined_by` is used to declare the struct's indices `m` and `n`. The attribute `lr::field` is used to annotate the struct field with a refinement type.

The type `RMat<m, n>` corresponds to a 2-dimensional matrix of size `m` × `n` that is implemented using a vector of vectors. The attribute `#[lr::refined_by]` on the `struct` definition specifies the type is indexed by two integers `m` and `n`. The attribute `#[lr::field]` is used to annotate the field `vec` with a refinement type. Flux will track and ensure that `vec` always has this type over the different `RMat` methods:

- `new` initializes a matrix of size `m` × `n` with all elements set to `0.0`, using `init_zeros` from fig. 2 to initialize each inner vector with zeros.
- `get` and `get_mut` are used to access the elements of the matrix using valid indices as arguments.

Flux can be used to ensure that the clients of such refined interfaces satisfy the specified invariants. In § 5 we use the `RVec` and `RMat` refined APIs to verify seven clients and conclude that verification requires less annotations and it is two times faster than program-logic based verifiers that use quantifiers.

Programs	P	::=	**fn** $main()$ **ret** k $\{F\}$ $\mid$ $D; P$	
Functions	D	::=	**fn** $f(\overline{x})$ **ret** k $\{F\}$	
Function Bodies	F	::=	**let** $x =$ **new**(n) **in** F	*let-bind*
		$\mid$	**letcont** $k(\overline{x}) = F$ **in** F	*continuation definition*
		$\mid$	$S; F$	*sequence*
		$\mid$	**if** p $\{F\}$ **else** $\{F\}$	*if-then-else*
		$\mid$	**jump** k $\langle\overline{r}\rangle$ $(\overline{x})$	*jump to continuation*
		$\mid$	**call** f $\langle\overline{r}\rangle$ $(\overline{x})$ **ret** k	*function call*
		$\mid$	**unpack**(x, v) **in** F	*unpacking*
Statements	S	::=	$p := rv$	*assignment*
		$\mid$	$p :=_n$ **copy** p	*copy memory*
		$\mid$	$p :=_n$ **move** p	*move memory*
R-values	rv	::=	c	*constants*
		$\mid$	&**mut** p	*mutable (re)borrow*
		$\mid$	$\&p$	*shared (re)borrow*
Places	p	::=	$x \mid *x$	*variable or derefence*
Constants	c	::=	**true** $\mid$ **false** $\mid 0, \pm 1, \ldots$	*booleans or integers*
Refinements	r	::=	$v \mid \ell \mid$ true $\mid$ false $\mid r + r \mid r < r \mid r = r \mid \ldots$	
Locations	ℓ	::=	$x \mid \rho$	*program or ghost variable*

Fig. 5. Syntax of Programs of λ_{LR}, with the refinement types related features highlighted.

3 DECLARATIVE TYPING

In this section, we introduce λ_{LR}, our formal version of Flux, which follows the Rust formalization in RustBelt [Jung et al. 2017]. By following their *semantic* approach, we can justify the refined interface of the (unsafe) implementation of RVec, extending their argument to prove that unsafe code can be safely encapsulated under a plain Rust interface. We begin (§ 3.1) by describing the syntax of λ_{LR}'s programs and types and then use it to define a syntactic type system (§ 3.2). Next (§ 3.3), we define an interpretation of λ_{LR}'s types into RustBelt's semantic model. Finally (§ 3.4), we state two soundness claims that respectively connect syntactic typing with the semantic model and runtime evaluation. The goal of this section is to formally present the parts of the system that affect refinement type checking. To center the discussion in this aspect, we leave out details related to *lifetimes* which we carry over from RustBelt. The complete definitions (including lifetimes) can be found in [Supplementary-Material 2022].

3.1 Syntax of λ_{LR}.

Figure 5 presents the syntax of λ_{LR} programs, which follows λ_{Rust}, the core language in RustBelt. It encodes an intermediate, continuation-passing style language that is much closer to Rust's Mid-level Intermediate Representation (MIR) than to surface Rust.

Variables The syntax distinguishes between program and refinement variables. Program variables (x), function names (f) and continuation variables (k) have runtime behavior. Refinement variables, which we write v or ρ, act as *ghost* code. We also use n to range over natural numbers.

Program & Function Definitions A program in λ_{LR} is a sequence of function definitions, ending with the special $main$ function which has no arguments. Functions are defined with the syntax **fn** $f(\overline{x})$ **ret** k $\{F\}$, where f is the function name, $\overline{x}$ the list of arguments and k the name of the return continuation, which can be called inside the *function body* F.

Types	τ	$:=$	$B\langle r\rangle$	*indexed type, e.g.,* $\mathbf{int}\langle 1\rangle$
		$\mid$	$\{v.\ B\langle v\rangle \mid r\}$	*existential type, e.g.,* $\{v.\ \mathbf{int}\langle v\rangle \mid 0 < v\}$
		$\mid$	$\mathbf{ptr}(\ell)$	*pointer to location* ℓ
		$\mid$	$\&_\mu\ \tau$	*borrowed reference (mutable or shared)*
		$\mid$	$\lightning_n$	*uninitialized memory of size* n
Base Types	B	$::=$	$\mathbf{int} \mid \mathbf{bool} \mid c$	*integers, booleans, or user-defined*
Modifier	μ	$::=$	$\mathbf{mut} \mid \mathbf{shr}$	*mutable or shared*
Sorts	σ	$:=$	$\mathbf{int} \mid \mathbf{bool} \mid \mathbf{loc}$	
Function signatures	s	$:=$	$\forall \overline{v:\sigma}.\ \mathbf{fn}(r; \overline{x}.\ \mathbf{T}) \to \rho_o.\ \mathbf{T}$	
Logical Environment	$\mathbf{\Delta}$	$:=$	$\emptyset \mid \mathbf{\Delta}, v:\sigma \mid \mathbf{\Delta}, r$	
Location contexts	$\mathbf{T}$	$:=$	$\emptyset \mid \mathbf{T}, \ell \mapsto \tau$	
Continuation contexts	$\mathbf{K}$	$:=$	$\emptyset \mid \mathbf{K}, k : \forall \overline{v:\sigma}.\ \mathbf{cont}(r; \overline{x}.\ \mathbf{T})$	
Global environments	$\mathbf{\Sigma}$	$:=$	$\emptyset \mid \mathbf{\Sigma}, f : s$	

Fig. 6. Syntax of Types and Environments in λ_{LR}.

Function Bodies In a function body, statements can be chained using $S; F$. Branching is supported using **if** p $\{F\}$ **else** $\{F\}$, where the condition is a *place* p, corresponding to a program variable x or a dereference $*x$. Unlike λ_{Rust}, program variables are not pure values and point to a location in memory, which we require to handle strong references. A variable pointing to uninitialized memory of size n can be declared using **let** $x = \mathbf{new}(n)$ **in** F. Functions can be called using **call** $f\langle\overline{r}\rangle(\overline{x})$ **ret** k, where the arguments are provided as a list of program variables following an A-normal form restriction which simplifies the definition of some typing rules [Jung et al. 2017]. More importantly, the call is annotated with a sequence of *refinements* r, which are drawn from an SMT-decidable logic. In our encoding, these include booleans, linear arithmetic, and (in)equality. These refinements are explicit in λ_{LR}, but are inferred by our implementation (§ 4.1) and not explicitly provided by the user. Control flow is handled through continuations, which are declared using **letcont** $k(\overline{x}) = F$ **in** F, and called using **jump** $k\langle\overline{r}\rangle(\overline{x})$. As with functions, calling a continuation requires a sequence of refinements which is inferred by our implementation. We allow continuations to be recursive to model loops. Finally, the command **unpack**(x, v) **in** F, which is also automatically introduced by our system, does nothing operationally but is required for type checking to connect the program variable x with the refinement variable v.

Statements & R-Values Statements in the language consist of three types of assignments. First, $p := rv$ assigns an *r-value* rv to a place p. An *r-value* rv can be used to assign a constant or a reference (mutable or shared). Next, the assignments $p :=_n$ **copy** p and $p :=_n$ **move** p are used to copy memory from one place to another. They have the same operational behavior, but different typing rules. We note that, unlike λ_{Rust}, we syntactically distinguish between statements and r-values to make the relevant type checking rules (§ 3.2) deterministic.

Types The syntax of types is shown in Figure 6. A base type B is either a primitive type (**int** or **bool**) or a user-defined type constructor c. The system can be extended with type constructors by providing a semantic interpretation as detailed in § 3.3. Refinement types extend base types with refinement expressions. We have two kinds as discussed in § 2.1: an indexed type $B\langle r\rangle$ indexes a base type B with a refinement r and an existential type $\{v.\ B\langle v\rangle \mid r\}$ represents a set of values constrained by a refinement r. Regular Rust references $\&_\mu\ \tau$ are qualified by a *modifier* μ which is either **mut** (for mutable references) or **shr** (for shared references)[2]. Strong references are modeled

[2]Crucially, references are also bound to a lifetime κ which we omit from the discussion for space reasons.

with the type $\mathbf{ptr}(\ell)$ representing a pointer to location ℓ. Finally, the type $\lightning_n$ describes uninitialized memory of size n.

Environments The system has four kinds of environments. The *logical environment* $\mathbf{\Delta}$ maps refinement variables to their sort ($v : \sigma$) and also contains predicates (r) constraining the variables in the context. The *location context* $\mathbf{T}$ describes ownership of locations, mapping them to their types. It is important to stress that the location context is substructural, modeling the fact that in Rust, only values of a type implementing `Copy` can we duplicated, which we write as τ `copy`. The *continuation context* $\mathbf{K}$ maps continuation names to their signatures. Similarly, the *global environment* $\mathbf{\Sigma}$ maps function names to their signature.

Function Signatures A function signature in λ_{LR} has the form $\forall \overline{v : \sigma}.\ \mathbf{fn}(r; \overline{x}.\ \mathbf{T}_i) \to \rho_{\text{o}}.\ \mathbf{T}_o$. It is parameterized by a list of refinement variables $\overline{v}$ each of a different sort σ, which can be either $\mathbf{int}$, $\mathbf{bool}$ or $\mathbf{loc}$. The input context $\mathbf{T}_i$ maps parameters $\overline{x}$ and locations in $\overline{v}$ to types. More importantly, it asserts that ownership of those locations has to be transferred to the function. The output context $\mathbf{T}_o$ has two purposes. First, it specifies the function's return type as a mapping to the *return location* ρ_{o}. Second, it maps input location in $\overline{v}$ to their updated type giving back their ownership. Finally, the refinement r is a precondition constraining the refinement variables $\overline{v}$.

As an example, we discuss the encoded signature of the function pop in fig. 3:

$$\forall n : \mathbf{int}, \rho : \mathbf{loc}.\ \mathbf{fn}(n > 0; x.\ \{\rho \mapsto \mathbf{vec}_\tau\langle n\rangle, x \mapsto \mathbf{ptr}(\rho)\}) \to \rho_{\text{o}}.\ \{\rho_{\text{o}} \mapsto \tau, \rho \mapsto \mathbf{vec}_\tau\langle n-1\rangle\}$$

The function is parameterized by two refinements an integer n and a location ρ. As declared in the input context, the function takes an argument x of type $\mathbf{ptr}(\rho)$, where the location ρ has type $\mathbf{vec}_\tau\langle n\rangle$. The indexed type $\mathbf{vec}_\tau\langle n\rangle$ encodes vectors of length n that contain elements of type τ, where $\mathbf{vec}_\tau$ is a user-defined constructor (§ 3.3). The output context states that the function returns a value of type τ and the location ρ is updated to hold a value of type $\mathbf{vec}_\tau\langle n-1\rangle$. Finally, the precondition specifies that the length n must be positive.

3.2 Declarative Judgements of λ_{LR}.

Figure 7 presents a selection of λ_{LR} main typing rules. Recall that the global environment $\mathbf{\Sigma}$ maps top level function names to their (user-provided) refined signature. Hence, a program P is well typed under $\mathbf{\Sigma}$ (T-Pg), if every function satisfies its signature in $\mathbf{\Sigma}$, which reduces to check if the function body is well typed under an adequate context (T-Def).

Well-typed Function Bodies The judgment $\mathbf{\Sigma}; \mathbf{\Delta} \mid \mathbf{K}; \mathbf{T} \vdash F$ checks if a function body is well typed under a global environment $\mathbf{\Sigma}$ which is not changed by the judgment. The context $\mathbf{\Delta}$ keeps track of refinement variables in scope, including locations. It is initialized with the input parameters in the function signature (T-Def). Additionally, $\mathbf{\Delta}$ contains constraints on the refinement variables which initially consist of the function precondition. The context $\mathbf{K}$ keeps track of continuations defined in the body, and it is initialized with the return continuation. The location context $\mathbf{T}$ maps locations (defined in $\mathbf{\Delta}$) to types, and it is initialized with the input context in the function signature.

The first four rules are standard as they do not directly manipulate refinements. Rule T-Let checks the body F of a let binding under an environment extended with the (freshly) defined location x. Ownership of the location is asserted by mapping it to uninitialized memory of size n in the location context $\mathbf{T}$. We assume that all variables inserted in the environments are α-renamed to be fresh. The rule T-Seq type checks the sequence of a statement S with the body F. Since the system is substructural, the rule must frame the location context splitting it into $\mathbf{T}_1$ and $\mathbf{T}$ of which only $\mathbf{T}_1$ is used to check the statement (additional structural rules are allowed by T-Inc). The statement judgment $\mathbf{\Delta} \mid \mathbf{T}_1 \vdash S \dashv \mathbf{T}_2$ consumes the context $\mathbf{T}_1$ and produces a

Well-typed Programs & Functions $\boxed{\Sigma \vdash P, \Sigma \vdash D}$

$$\frac{\forall D_i \in \overline{D}.\, \Sigma \vdash D_i}{\Sigma \vdash \overline{D}}\ \text{T-Pg} \qquad \frac{\begin{array}{c}\Sigma(f) = \forall \overline{v:\sigma}.\, \mathbf{fn}(r; \overline{x}.\, \mathbf{T}_i) \to \rho_o.\, \mathbf{T}_o \\ \Sigma; \overline{x} : \mathbf{loc}, \overline{v:\sigma}, r \mid \mathbf{T}_i; k : \mathbf{cont}(\mathbf{true}; \rho_o.\, \mathbf{T}_o) \vdash F\end{array}}{\Sigma \vdash \mathbf{fn}\ f(\overline{x})\ \mathbf{ret}\ k\ \{F\}}\ \text{T-Def}$$

Well-typed Function Bodies $\boxed{\Sigma; \Delta \mid \mathbf{K}; \mathbf{T} \vdash F}$

$$\frac{\Delta, x : \mathbf{loc} \mid \mathbf{T}, x \mapsto \not\downarrow_n; \mathbf{K} \vdash F}{\Delta \mid \mathbf{K}; \mathbf{T} \vdash \mathbf{let}\ x = \mathbf{new}(n)\ \mathbf{in}\ F}\ \text{T-Let} \qquad \frac{\Delta \mid \mathbf{T}_1 \vdash S \dashv \mathbf{T}_2 \qquad \Delta \mid \mathbf{K}; \mathbf{T}_2, \mathbf{T} \vdash F}{\Delta \mid \mathbf{K}; \mathbf{T}_1, \mathbf{T} \vdash S; F}\ \text{T-Seq}$$

$$\frac{\begin{array}{c}\Delta \mid \mathbf{K}; \mathbf{T}_1 \vdash F \\ \Delta \vdash \mathbf{T} \Rightarrow \mathbf{T}_1\end{array}}{\Delta \mid \mathbf{K}; \mathbf{T} \vdash F}\ \text{T-Inc} \qquad \frac{\begin{array}{c}\Delta, \overline{v:\sigma}, \overline{x} : \mathbf{loc}, r \mid \mathbf{K}, k : \forall \overline{v:\sigma}.\, \mathbf{cont}(r; \overline{x}.\, \mathbf{T}'); \mathbf{T}' \vdash F_k \\ \Delta \mid \mathbf{K}, k : \forall \overline{v:\sigma}.\, \mathbf{cont}(r; \overline{x}.\, \mathbf{T}'); \mathbf{T} \vdash F\end{array}}{\Delta \mid \mathbf{K}; \mathbf{T} \vdash \mathbf{letcont}\ k(\overline{x})\ =\ F_k\ \mathbf{in}\ F}\ \text{T-Cont}$$

$$\frac{\sigma = \mathtt{sort}(B) \qquad \Delta, v : \sigma, r[v/b] \mid \mathbf{K}; \mathbf{T}, \ell \mapsto B\langle v \rangle \vdash F}{\Delta \mid \mathbf{K}; \mathbf{T}, \ell \mapsto \{b.\, B\langle b \rangle \mid r\} \vdash \mathbf{unpack}(x, v)\ \mathbf{in}\ F}\ \text{T-unpack}$$

$$\frac{\begin{array}{c}\mathbf{K}(k) = \mathbf{cont}(\mathbf{true}; z.\, \mathbf{T}_k) \qquad \Sigma(f) = \forall \overline{v:\sigma}.\, \mathbf{fn}(r; \overline{y}.\, \mathbf{T}_i) \to \rho_o.\, \mathbf{T}_o \\ \theta = [\overline{r}/\overline{v}][\overline{x}/\overline{y}] \qquad \overline{\Delta \vdash r : \sigma} \qquad [\![\Delta \vdash \theta \cdot r]\!] \\ \Delta \vdash \mathbf{T}_1 \Rightarrow \theta \cdot \mathbf{T}_i \qquad \Delta \vdash \mathbf{T}_2, \theta \cdot \mathbf{T}_o \Rightarrow \mathbf{T}_k[\rho_o/z]\end{array}}{\Sigma; \Delta \mid \mathbf{K}; \mathbf{T}_1, \mathbf{T}_2 \vdash \mathbf{call}\ f\langle \overline{r} \rangle(\overline{x})\ \mathbf{ret}\ k}\ \text{T-call}$$

$$\frac{\begin{array}{c}\mathbf{K}(k) = \forall \overline{v:\sigma}.\, \mathbf{cont}(r'; \overline{y}.\, \mathbf{T}') \\ \overline{\Delta \vdash r : \sigma} \qquad [\![\Delta \vdash \theta \cdot r']\!] \\ \theta = [\overline{x}/\overline{y}][\overline{r}/\overline{v}] \qquad \Delta \vdash \mathbf{T} \Rightarrow \theta \cdot \mathbf{T}'\end{array}}{\Delta \mid \mathbf{K}; \mathbf{T} \vdash \mathbf{jump}\ k\langle \overline{r} \rangle(\overline{x})}\ \text{T-Jump} \qquad \frac{\begin{array}{c}\mathbf{T} \vdash p \overset{\text{cp}}{\rightsquigarrow} \mathbf{bool}\langle r \rangle \\ \Delta, r \mid \mathbf{K}; \mathbf{T} \vdash F_1 \\ \Delta, \neg r \mid \mathbf{K}; \mathbf{T} \vdash F_2\end{array}}{\Delta \mid \mathbf{K}; \mathbf{T} \vdash \mathbf{if}\ p\ \{F_1\}\ \mathbf{else}\ \{F_2\}}\ \text{T-If}$$

Well-typed Statements $\boxed{\Delta \mid \mathbf{T}_i \vdash S \dashv \mathbf{T}_o}$

$$\frac{\begin{array}{c}\mathbf{T}_i \vdash p_r \overset{\text{cp}}{\rightsquigarrow} \tau \qquad \mathtt{size}(\tau) = n \\ \Delta \mid \mathbf{T}_i \vdash p_l \leftsquigarrow \tau \dashv \mathbf{T}_o\end{array}}{\Delta \mid \mathbf{T}_i \vdash p_l :=_n \mathbf{copy}\ p_r \dashv \mathbf{T}_o}\ \text{T-copy} \qquad \frac{\begin{array}{c}\mathbf{T}_i \vdash p_r \overset{\text{mv}}{\rightsquigarrow} \tau \dashv \mathbf{T} \qquad \mathtt{size}(\tau) = n \\ \Delta \mid \mathbf{T} \vdash p_l \leftsquigarrow \tau \dashv \mathbf{T}_o\end{array}}{\Delta \mid \mathbf{T}_i \vdash p_l :=_n \mathbf{move}\ p_r \dashv \mathbf{T}_o}\ \text{T-move}$$

$$\frac{\Delta \mid \mathbf{T}_i \vdash rv \rightsquigarrow \tau \dashv \mathbf{T} \qquad \Delta \mid \mathbf{T} \vdash p \leftsquigarrow \tau \dashv \mathbf{T}_o}{\Delta \mid \mathbf{T}_i \vdash p := rv \dashv \mathbf{T}_o}\ \text{T-assign}$$

Fig. 7. Selected rules for the main checking judgments.

new context $\mathbf{T}_2$, which is combined with $\mathbf{T}$ to check F. Since continuations can be recursive, rule T-Cont checks the body F_k by guessing an appropriate signature for k. (In § 4.2 we explain how Flux algorithmically infers k's signature.) Further, the logical environment is extended with the continuation parameters and precondition.

The next four rules have a more interesting interaction with refinements. The rule T-call checks a function call **call** $f\langle\overline{r}\rangle(\overline{x})$ **ret** k. It retrieves the functions signature $\forall\overline{v:\sigma}.\ \mathbf{fn}(r;\overline{y}.\ \mathbf{T}_i) \to \rho_o.\ \mathbf{T}_o$ from the global environment and uses it to perform four checks. First, the refinement arguments $\overline{r}$ must have the same sort as the refinement parameters. This check is performed by $\mathbf{\Delta} \vdash r : \sigma$ which is a standard unrefined typing judgment. Second, the function's precondition r must hold under the logical environment $\mathbf{\Delta}$, which is checked by the judgement $[\![\mathbf{\Delta} \vdash r]\!]$. This judgment ensures first order logic validity of r under the information captured by $\mathbf{\Delta}$. The last two premises of the rule respectively check compatibility with the function's input ($\mathbf{T}_i$) and output ($\mathbf{T}_o$) contexts. To perform these checks the function's call context is split into $\mathbf{T}_1$ and $\mathbf{T}_2$, $\mathbf{T}_1$ should be included in $\mathbf{T}_i$ after the proper substitution defined by θ, while $\mathbf{T}_2$ combined with the substituted output context ($\theta \cdot \mathbf{T}_o$) should be included in the continuation context $\mathbf{T}_k$. Similarly, the rule T-Jump checks **jump** $k\langle\overline{r}\rangle(\overline{x})$ by retrieving the type of the continuation k, and then ensuring that the refinement arguments $\overline{r}$ are of the required sort, that the precondition r' is satisfied, and that the continuation's context $\mathbf{T}'$ (after substitution) includes the jumping context $\mathbf{T}$. The rule T-If checks **if** p $\{F_1\}$ **else** $\{F_2\}$ and is path sensitive. Concretely, F_1 and F_2 are respectively checked in a logical environment extended with the path condition r and its negation. To extract the path condition the rule uses the type copying judgment $\mathbf{T} \vdash p \overset{\text{cp}}{\rightsquigarrow} \tau$ that follows pointers in a place to retrieve its type. Finally, the rule T-unpack is used to open an existential $\{b.\ B\langle b\rangle \mid r\}$. It extends the logical environment with the variable v, asserting the constraint r, and updates ℓ to point to $B\langle v\rangle$.

Well-typed Statements Typing of statements has little interaction with refinements. Rule T-copy checks copy assignments using the auxiliary type copying judgment $\mathbf{T}_i \vdash p_r \overset{\text{cp}}{\rightsquigarrow} \tau$ to look up the type τ of the place p_r ensuring it satisfies τ copy, this includes integer, booleans and shared references. It then generates an output context using the type writing judgment $\mathbf{\Delta} \mid \mathbf{T}_i \vdash p_l \leftsquigarrow \tau \dashv \mathbf{T}_o$ which updates the type associated to p_l. The rule T-move is similar, except that it uses the type moving judgment $\mathbf{T}_i \vdash p_r \overset{\text{mv}}{\rightsquigarrow} \tau \dashv \mathbf{T}_o$ which does not require τ copy but outputs an updated context where p_r is mapped to uninitialized memory. Finally, T-assign uses the judgment $\mathbf{\Delta} \mid \mathbf{T}_i \vdash rv \rightsquigarrow \tau \dashv \mathbf{T}$ to synthesize a type for the right-hand side which can update the context to reflect the effects of borrowing. All the details are spelled out in [Supplementary-Material 2022].

Context Inclusion and Subtyping We finish this section by discussing subtyping which has an important interaction with refinements. Figure 8 presents selected rules of the context inclusion and subtyping judgments. The first three rules of context inclusion are structural and allow permutation, weakening and framing. The rule C-Sub reduces inclusion to subtyping.

The main form of subtyping supported in Rust involves relationships between lifetimes which carries over to λ_{LR}, but we also extend subtyping to refinements. The rule S-RType establishes that $B\langle r_1\rangle$ is a subtype of $B\langle r_2\rangle$ if $\mathbf{\Delta}$ logically entails the equality $r_1 = r_2$. The rule S-Unpack allows unpacking an existential when it appears on the left of the judgment. Dually, when an existential appears on the right we can reduce the subtyping $\mathbf{\Delta} \vdash B\langle r_1\rangle \preccurlyeq \{v.\ B\langle v\rangle \mid r_2\}$ to the validity of r_2 by picking r_1 as a witness for the variable v. Finally, the rules S-Bor-Shr and S-Bor-Mut witness the variance of references. Shared references are covariant while mutable references have to be checked both covariantly and contravariantly.

Context inclusion $\boxed{\Delta \vdash \mathbf{T} \Rightarrow \mathbf{T}}$

$$\frac{\mathbf{T}' \text{ is a permutation of } \mathbf{T}}{\Delta \vdash \mathbf{T} \Rightarrow \mathbf{T}'}\ \text{C-Perm} \qquad \frac{}{\Delta \vdash \mathbf{T}, \mathbf{T}' \Rightarrow \mathbf{T}}\ \text{C-Weaken}$$

$$\frac{\Delta \vdash \mathbf{T}_1 \Rightarrow \mathbf{T}_2}{\Delta \vdash \mathbf{T}, \mathbf{T}_1 \Rightarrow \mathbf{T}, \mathbf{T}_2}\ \text{C-Frame} \qquad \frac{\Delta \vdash \tau_1 \preccurlyeq \tau_2}{\Delta \vdash \ell \mapsto \tau_1 \Rightarrow \ell \mapsto \tau_2}\ \text{C-Sub}$$

Subtyping $\boxed{\Delta \vdash \tau \preccurlyeq \tau}$

$$\frac{}{\Delta \vdash \tau \preccurlyeq \tau}\ \text{S-Refl} \qquad \frac{\Delta \vdash \tau_1 \preccurlyeq \tau_2 \qquad \Delta \vdash \tau_2 \preccurlyeq \tau_3}{\Delta \vdash \tau_1 \preccurlyeq \tau_3}\ \text{S-Trans} \qquad \frac{[\![\Delta \vdash r_1 = r_2]\!]}{\Delta \vdash B\langle r_1 \rangle \preccurlyeq B\langle r_2 \rangle}\ \text{S-RType}$$

$$\frac{\Delta, v : \mathtt{sort}(B), r \vdash B\langle v \rangle \preccurlyeq \tau}{\Delta \vdash \{v.\ B\langle v \rangle \mid r\} \preccurlyeq \tau}\ \text{S-Unpack} \qquad \frac{[\![\Delta \vdash r_2[r_1/v]]\!]}{\Delta \vdash B\langle r_1 \rangle \preccurlyeq \{v.\ B\langle v \rangle \mid r_2\}}\ \text{S-Exists}$$

$$\frac{\Delta \vdash \tau_1 \preccurlyeq \tau_2}{\Delta \vdash \&_{\mathbf{shr}}\ \tau_1 \preccurlyeq \&_{\mathbf{shr}}\ \tau_2}\ \text{S-Bor-Shr} \qquad \frac{\Delta \vdash \tau_1 \preccurlyeq \tau_2 \qquad \Delta \vdash \tau_2 \preccurlyeq \tau_1}{\Delta \vdash \&_{\mathbf{mut}}\ \tau_1 \preccurlyeq \&_{\mathbf{mut}}\ \tau_2}\ \text{S-Bor-Mut}$$

Fig. 8. Context Inclusion & Subtyping (selected rules).

3.3 Semantic Model of λ_{LR} Types.

Next, we describe an interpretation of refinement types in RustBelt's semantic model, which we use to define a soundness claim for λ_{LR} in § 3.4.

RustBelt's Model In RustBelt each type τ is associated to an *ownership predicate* $[\![\tau]\!](\overline{v})$ that semantically models what it means to own an object of type τ[3]. As the data in Rust is typically laid out in memory and spans multiple locations, the predicate takes a list of values. The predicate is written in Iris [Jung et al. 2018], a higher-order concurrent separation logic formalized in Coq, which facilitates the complex reasoning about Rust's ownership mechanism. Concretely, the ownership predicate has type list(*Val*) $\to$ *iProp*, where *Val* is the type of values in the language and *iProp* is the type of Iris propositions. As an example, we present the semantic interpretation of (unrefined) booleans as originally defined in RustBelt.

$$[\![\mathbf{bool}]\!](\overline{v}) \doteq \overline{v} = [\mathbf{true}] \vee \overline{v} = [\mathbf{false}]$$

In other words, a boolean value can only be the singleton list, and that list contains either **true** or **false**.

Modeling Refinement Types Before defining the interpretation of refinement types we need some initial setup. Since refinements can contain variables, we parameterize the interpretation of types with respect to a substitution γ that maps refinement variables to closed expressions, *i.e.,* expressions that are well sorted under the empty environment:

$$\gamma = \{(v_i \leftarrow r_i) \mid \exists \sigma_i. \emptyset \vdash r_i : \sigma_i\}$$

[3]Each type is also associated to a *sharing predicate* and a *size predicate*, but we omit the details for space reasons.

With a substitution in place, we can define an interpretation for refinements $[\![r]\!]_\gamma$ as the "standard" encoding of our SMT-decidable fragment into Iris. The interpretation for refinement types follows:

$$[\![\mathbf{int}\langle r\rangle]\!]_\gamma(\overline{v}) \doteq \overline{v} = [[\![r]\!]_\gamma] * [\![r]\!]_\gamma \in \mathbb{Z}$$
$$[\![\mathbf{bool}\langle r\rangle]\!]_\gamma(\overline{v}) \doteq \overline{v} = [[\![r]\!]_\gamma] * [\![r]\!]_\gamma \in \{\mathbf{true}, \mathbf{false}\}$$
$$[\![\mathbf{ptr}(\ell)]\!]_\gamma(\overline{v}) \doteq \overline{v} = [[\![\ell]\!]_\gamma] * [\![\ell]\!]_\gamma \in \mathit{Loc}$$

Integers and booleans are interpreted as lists containing a single element that equals the interpretation of the index. Crucially, the interpretation of the index must have the appropriate sort. Similarly, a pointer $\mathbf{ptr}(\ell)$ is a singleton list containing a location. The interpretation of existentials is inductive:

$$[\![\{v.\ B\langle v\rangle \mid r\}]\!]_\gamma(\overline{v}) \doteq \exists w. [\![B\langle v\rangle]\!]_{\gamma[v\leftarrow w]}(\overline{v}) * [\![r]\!]_{\gamma[v\leftarrow w]}$$

An existential $\{v.\ B\langle v\rangle \mid r\}$ asserts the existence of a pure value w satisfying the (interpretation of the) constraint r and delegates ownership to $B\langle v\rangle$.

Modeling Vectors The key feature in RustBelt is to justify that the system can be extended with libraries that encapsulate unsafe code under well-typed abstraction. We use the same argument to justify the interface for `RVec` discussed in § 2.3 by providing a semantic interpretations of vectors.

In Rust a vector has three fields: (1) a capacity corresponding to the amount of allocated memory, (2) a length corresponding to the current number of elements in the vector, which must not exceed the capacity, and (3) a pointer to the actual memory. Thus, we define the interpretation of vectors as follows:

$$[\![\mathbf{vec}_\tau\langle r\rangle]\!]_\gamma(\overline{v}) \doteq \exists c, n, \ell.\, \overline{v} = [c, n, \ell] * n < c * n = [\![r]\!]_\gamma * [\![r]\!]_\gamma \in \mathbb{N} * \text{OwnVec (c, n, } \ell, \tau)$$

Since the language does not support polymorphism we follow RustBelt and define the predicate by quantifying over all types τ at the meta-level. A vector is then a list of three values, a capacity c, a length n and a pointer ℓ. More importantly, the length n is equal to the index r, which must be a natural number. Finally, OwnVec is an Iris predicate which asserts ownership of the memory pointed to by ℓ and that the first n positions contain elements of type τ.

3.4 Soundness of λ_{LR}.

Finally, we state two soundness claims for λ_{LR}. First (§ 3.4.1), denotational soundness (Theorem D.1) states that if a program syntactically type checks, then it obeys the semantic model of types (of § 3.3). Second (§ 3.4.2), we define the operational semantics of λ_{LR} to express in Theorem D.2 that if a program type checks, it will not get stuck at runtime.

3.4.1 Model of Judgments & Denotational Soundness. Based on the modeling of λ_{LR} types (§ 3.3) we define the semantic interpretations of the syntactic typing environments and judgments of § 3.2. Below we define the interpretation of the logical environment $\mathbf{\Delta}$ and location context $\mathbf{T}$.

$$\begin{array}{lll} \emptyset \in [\![\emptyset]\!] & \doteq \text{True} & [\![\emptyset]\!]_\gamma \quad \doteq \text{True} \\ \gamma[r \leftarrow v] \in [\![\mathbf{\Delta}, v : \sigma]\!] & \doteq \gamma \in [\![\mathbf{\Delta}]\!] \wedge \emptyset \models r : \sigma & [\![\mathbf{T}, \ell \mapsto \tau]\!]_\gamma \doteq [\![\mathbf{T}]\!]_\gamma * \exists \overline{v}. [\![\ell]\!]_\gamma \mapsto \overline{v} * [\![\tau]\!]_\gamma(\overline{v}) \\ \gamma \in [\![\mathbf{\Delta}, r]\!] & \doteq \gamma \in [\![\mathbf{\Delta}]\!] \wedge \gamma \models [\![r]\!]_\gamma & \end{array}$$

On the left, a substitution γ belongs in the interpretation of $\mathbf{\Delta}$ when all the mappings on γ respect the sorts of $\mathbf{\Delta}$ and satisfy $\mathbf{\Delta}$'s refinements. On the right, the interpretation of the location context $\mathbf{T}$ defines an *iProp* stating that for each mapping $\ell \mapsto \tau$ in the context, there exists a list of values $\overline{v}$ so that the interpretation of ℓ points to $\overline{v}$ and the values $\overline{v}$ are owned at type τ. The definition uses the points-to connective of separation logic to assert ownership of the location.

Using the interpretations of environments, we define the semantic judgements of λ_{LR}. As an example, we present the semantic typing of programs and function definitions.

$$\begin{aligned}\Sigma \models D; P &\doteq \Sigma \models D * \Sigma \models P \\ \Sigma \models \textbf{fn}\ f(\overline{x})\ \textbf{ret}\ k\ \{F\} &\doteq (f, \forall \overline{v:\sigma}.\ \textbf{fn}(r; \overline{x}.\ \mathbf{T}_i) \to \rho_o.\ \mathbf{T}_o) \in \Sigma \\ &\quad * \forall \gamma \in [\![\overline{x} : \textbf{loc}, \overline{v:\sigma}, r]\!].\\ &\quad \{[\![\Sigma]\!]_\gamma * [\![\mathbf{T}_i]\!]_\gamma * [\![k : \textbf{cont}(\textbf{true}; \rho_o.\ \mathbf{T}_o)]\!]_\gamma\} F \{\text{True}\}\end{aligned}$$

A program is accepted by the semantic judgment when all function definitions are accepted. To define the interpretation of the semantic judgment for a function definition $\textbf{fn}\ f(\overline{x})\ \textbf{ret}\ k\ \{F\}$, we use hoare triples. From the global environment Σ we extract the function's signature and (as in the syntactic rule T-Def in fig. 7) derive the function's logical environment $\Delta = \{\overline{x} : \textbf{loc}, \overline{v:\sigma}, r\}$, continuation context $\mathbf{K} = \{k : \textbf{cont}(\textbf{true}; \rho_o.\ \mathbf{T}_o)\}$, and input location contexts $\mathbf{T}_i$. The semantic judgment states that for each substitution γ that belongs in the interpretation of f's original logical environment (*i.e.,* $\gamma \in \Delta$), the function body F has the pre-conditions defined by the interpretations of the three input environments Σ, $\mathbf{T}_i$, and $\mathbf{K}$. The post-condition is just True following the treatment of full program execution in RustBelt.

Denotational soundness states that if a program syntactically type checks, then it also respects the semantic judgment.

Theorem 3.1 (Denotational Soundness). *If $\Sigma \vdash P$, then $\Sigma \models P$.*

3.4.2 Operational Semantics & Stuck Freedom. Finally, we want to state that well-typed programs, cannot get stuck. To do so, we define the operational semantics of λ_{LR} by translation into the core calculus of RustBelt. That is, all the primitives of λ_{LR} are defined as derived forms in the core calculus. This core language is a lambda calculus equipped with primitive values and pointer arithmetic. We present the definition of some derived form to illustrate that refinements do not affect runtime evaluation.

$$\begin{aligned}\textbf{call}\ f\langle r\rangle(\overline{e})\ \textbf{ret}\ k &\doteq f([k] +\!\!+ \overline{e}) \\ \textbf{jump}\ k\langle r\rangle(\overline{e}) &\doteq k\ (\overline{e}) \\ \textbf{unpack}(x, v)\ \textbf{in}\ F &\doteq F\end{aligned}$$

A function call passes the return continuation k as an extra argument. The definition of **jump**, simply calls the continuation. In both cases the refinement arguments r, which only serve for refinement type checking, are ignored. Similarly, the unpack primitive, is a non operation at runtime.

To evaluate λ_{LR} programs, we simply use the operational semantics of the core calculus. A program reaches a stuck state when none of the operational rules apply. Theorem D.2 states that if a λ_{LR} program type checks, a stuck state will not be reached.

Theorem 3.2 (Stuck Freedom). *If $\Sigma \models P$, then the execution of $main$ with the default continuation, i.e., $main(x.x)$, will not end in a stuck state.*

4 ALGORITHMIC VERIFICATION

Flux implements the type checking rules of § 3 as a Rust compiler plugin, adding an extra analysis step to the compiler pipeline (similar to the Clippy[4] linter). As a compiler plugin, Flux operates on compiled Rust programs, which has two major benefits. First, the programs satisfy Rust's safety requirements, which are assumed by our analysis. For example, Flux assumes that every

[4] https://github.com/rust-lang/rust-clippy

access through a reference satisfies Rust's *borrow-checker* rules. Second, the Rust compiler generates the typed intermediate representation of compiled programs, which is passed to Flux. Concretely, Flux performs the analysis on Rust Mid-level Intermediate Representation (MIR) which is extracted from the compiler combined with (non refined) type information. MIR is a control-flow graph (CFG) representation, unlike our core calculus λ_{LR} that is in continuation passing style (CPS), but this is not a challenge, since both representations are easy to relate via the correspondence defined by Appel [2007].

Yet, there were three main challenges in implementing Flux as the core system presented in § 3. First (§ 4.1), the syntax of λ_{LR} has explicit refinement annotations (the highlighted parts of fig. 5). that do not appear in Rust's MIR. Second (§ 4.2), the λ_{LR} judgements have rules that require inference, *e.g.,* the rule T-Cont of fig. 7 guesses the continuation's context. Finally (§ 4.3), Flux supports polymorphic types which is crucial for ergonomic specification and verification, but require the instantiation of refinement types which is, in general, challenging to implement. We explain how we addressed these three challenges via the liquid types inference framework [Cosman and Jhala 2017; Rondon et al. 2008].

4.1 Refinement Annotations

Flux, following the essence of refinement typing, does not modify the syntax of Rust programs, but permits refined type signatures. So, the users should explicitly declare the refined type signatures of λ_{LR} (using the user-friendly syntax of § 2), but the **unpack** instructions and the refinement parameters of function calls and continuation jumps are automatically inferred by the system.

The inference of **unpack** instructions happens on-the-fly. The MIR code analyzed by Flux does not have explicit **unpack** instructions. Instead, Flux introduces a fresh refinement variable as soon as an existential type goes into the context. In this way it can keep the context normalized, which is required for the instantiation of refinement parameters.

The refinement parameter instantiation is performed by a syntax-directed heuristic. Intuitively, Flux uses the `@n` syntax to connect each refinement parameter with a concrete function argument. For instance, consider the function `is_pos` from § 1 which refines its argument using `i32<@n>`. In the call `is_pos(1)`, we can match the type `i32<1>` with `i32<@n>` and instantiate `n` to `1`.

This strategy performs well in practice, and it is sufficient to instantiate the parameters in all the benchmarks in our evaluation (§ 5), but it is not complete, for example it cannot infer a parameter declared inside a polymorphic argument. In case of instantiation failure Flux exits with an error and the user can, as a fallback option, provide additional arguments to simplify the instantiation. For example, the function `normalize_centers` in fig. 2 has to take an additional parameter to declare `@n` because Flux would not be able to instantiate it inside the inner `RVec`.

4.2 Refinement Inference

The rule T-Cont requires guessing (*a.k.a.,* inferring) a signature for the continuation that, combined with the (also non-algorithmic) context inclusion judgement, is used to check loop invariants. In Flux such loop invariants are encoded as logical invariants in the continuation contexts and are inferred in a *two phase* analysis. As an example, consider the function `init_zeros` in fig. 2 of § 2. In this function, the header of the loop corresponds to continuation, let's call it k, and it is associated to a context $\mathbf{T}_k$. The goal is to infer this context.

In the first phase of the analysis, Flux collects all the join points of the continuation, *i.e.,* all the contexts that jump to k. In the `init_zeros` example, the loop is entered under two different

contexts, the initial context $\mathbf{T}_0$ and the loop body $\mathbf{T}_i$:

$$\begin{array}{rcl} \mathbf{T}_0 & \doteq & \{\texttt{n} \mapsto \texttt{usize<}v\texttt{>}, \texttt{i} \mapsto \texttt{i32<0>}, \texttt{vec} \mapsto \texttt{RVec<f32,0>}\} \\ \mathbf{T}_i & \doteq & \{\texttt{n} \mapsto \texttt{usize<}v\texttt{>}, \texttt{i} \mapsto \texttt{i32<}j+1\texttt{>}, \texttt{vec} \mapsto \texttt{RVec<f32,}j+1\texttt{>}\} \end{array}$$

Where `n`, `i`, and `vec` are the locations in scope. The variable v indexing the type of `n` is in scope for the entire function and since `n` is not mutated through the loop its type is the same in both contexts. The refinement variable j signifies the value of `i` in the previous iteration, and it is out of scope outside the body of loop. The goal of inference is to construct a continuation context that includes both $\mathbf{T}_0$ and $\mathbf{T}_1$ and it is strong enough to prove that the length of `vec` equals `n`. By syntactic unification, the first phase of the analysis will construct the following continuation context.

$$k : \forall b, c.\, \mathbf{cont}(\kappa(b, c); \mathbf{T}_k), \text{ where } \mathbf{T}_k \doteq \{\texttt{n} \mapsto \texttt{usize<}v\texttt{>}, \texttt{i} \mapsto \texttt{i32<}b\texttt{>}, \texttt{vec} \mapsto \texttt{RVec<f32,}c\texttt{>}\}$$

That is, the derived context preserves the matching parts (here the index of `n`), while for the rest (here the indices of `i` and `vec`) fresh logical names are generated (here b and c). Importantly, the fresh names can be related by the *refinement variable* κ, *i.e.,* an unknown predicate capturing dependencies between logical values whose value will be decided using liquid types inference in the next phase.

In the second phase the type checking rules of § 3 are applied. These rules check context inclusion at each join point (rule T-Jump) which, in turn, reduces to semantic subtyping (rule C-Sub) that essentially performs validity checking of logical conditions. In our example, the two continuation calls lead to the following context inclusion checks.

$$\begin{array}{lcl} \mathbf{\Delta}_0 \vdash \mathbf{T}_0 \Rightarrow \mathbf{T}_k[0/b][0/c] & \Rightarrow & \kappa(0,0) \\ \mathbf{\Delta}_i \vdash \mathbf{T}_i \Rightarrow \mathbf{T}_k[j+1/b][j+1/c] & \Rightarrow & \kappa(j+1, j+1) \end{array}$$

For typechecking to succeed, the context inclusion judgements on the left should hold, which (using the typing rules) reduce to the logical conditions on the right. These conditions allow liquid inference to find the solution $\kappa(b, c) := b = c$ for the unknown variable κ, which is strong enough to verify the top level specification of `init_zeros`, *i.e.,* that at the exit of the loop (when $v = b$), the length of `vec` is equal to `n`.

4.3 Polymorphic Instantiation

Flux further infers invariants over elements of type constructors by instantiating the polymorphic types of the constructors by existential types with unknown predicates and solving them using liquid inference. For example, consider the following function that creates and returns a vector with one element.

```
1 #[lr::sig(fn() -> RVec<i32{v : v > 0}>)]
2 fn make_vec() -> RVec<i32> {
3     let vec = RVec::new();    // vec ↦ RVec<i32{ν : κ1(ν)}, 0>
4     RVec::push(&mut vec, 42); // vec ↦ RVec<i32{ν : κ2(ν)}, 1>
5     vec
6 }
```

The comments show the type of `vec` after each statement. The call to `new` needs to instantiate the parameter `T` in the return type `RVec<T, 0>` . Flux extracts from the Rust compiler that `T` is instatiated by an `i32` but the refinement is unknown. Thus, `T` gets instantiated by the template `i32{`$\nu : \kappa_1(\nu)$`}` where κ_1 is a fresh refinement variable denoting an unknown predicate. Similarly, the call to `push` also requires instantiating its polymorphic parameter, so the template

	Flux			Prusti			
	LOC	**Spec**	**Time (s)**	**LOC**	**Spec**	**Annot**	**Time (s)**
Library							
`RVec`	47	22	-	45	29	-	-
`RMat`	30	8	1.4	41	16	-	-
Total	77	30	1.4	86	45	-	-
Benchmark							
`bsearch`	25	1	0.8	25	0	1	2.3
`dotprod`	12	1	0.7	12	1	1	2.1
`fft`	180	17	3.1	188	22	24	240.6
`heapsort`	39	2	1.1	37	5	9	7.7
`simplex`	124	10	2.7	125	25	8	21.6
`kmeans`	94	12	2.7	87	37	10	21.1
`kmp`	48	2	1.9	49	4	7	9.9
Total	522	45	13	523	94	60	304.2

Table 1. Experimental results comparing Flux and Prusti. **LOC** is the number of lines of Rust *source code*, **Spec** is the number of lines for function *specifications*, **Annot** is the amount of lines for user-specified *loop invariants*, and **Time (s)** is the time in seconds required to verify the code (trusted code does not have an associated time).

`i32`$\{\nu : \kappa_2(\nu)\}$ gets created. Type checking the function with these templates generates three logical constraints:

$$1) \quad \kappa_1(\nu) \Rightarrow \kappa_2(\nu) \qquad 2) \quad \nu = 42 \Rightarrow \kappa_2(\nu) \qquad 3) \quad \kappa_2(\nu) \Rightarrow \nu > 0$$

The first two constraints correspond to subtyping for the first and second arguments to `push`; the third relates the type of `vec` to the output type. Using liquid inference, Flux solves both unknows to $\nu > 0$ (*i.e.*, $\kappa_1(\nu) := \kappa_2(\nu) := \nu > 0$) which is a predicate strong enough to verify the specification of `make_vec`.

5 EVALUATION

Next, we present an empirical evaluation of the benefits of Flux's refinement type-based verification to classic program logic-based approaches as embodied in Prusti [Astrauskas et al. 2019], a state-of-the-art program logic based verifier for Rust that also exploits the implicit *capability* information present in Rust's type system to reduce the verification overhead. Prusti supports far more expressive specifications than Flux allowing users to verify various forms of functional correctness properties (*e.g.*, sorted-ness) via pure predicates. However, we show that for many common and important use-cases, program-logic based methods make verification unnecessarily cumbersome. In particular, our evaluation focuses on three dimensions for comparison: do types (§ 5.1) facilitate *faster* verification? (§ 5.2) enable *compact* specifications? (§ 5.3) require *fewer* annotations?

Benchmarks We compare Flux and Prusti on a set of vector-manipulating benchmark programs drawn from the literature [Astrauskas et al. 2019; Rondon et al. 2008], which implement loop-heavy algorithms over the `RVec` and `RMat` collection libraries discussed in § 2.3. In each case, the verification goal is to prove the safety of vector accesses for the program. The benchmarks are listed in table 1. The first five benchmarks are ported from the Dsolve project [Rondon et al. 2008], a refinement type system for Ocaml. These include implementations of: Binary Search (`bsearch`),

computing the Dot Product of two vectors (`dotprod`), Fast Fourier Transform (`fft`), Heap Sort (`heapsort`), and the Simplex algorithm for Linear Programming (`simplex`). The last two benchmarks are implementations of the k-means clustering algorithm (`kmeans`), and the Knuth-Morris-Pratt string-searching algorithm (`kmp`). These two were chosen to highlight the ability of Flux to express quantified invariants via polymorphism. In each case, we first implemented and verified the code in Flux, and then tried to replicate it as closely as possible in Prusti.

Setup We ran all the experiments on a commodity laptop running Windows 11 with 16GB of memory and a quad core Intel(R) Core(TM) i7-1065G7 CPU @ 1.50GHz. We used the following versions of the software required to run Prusti: (1) Prusti Release v-2021-11-22-1738, (2) Z3 v4.8.9, (3) jdk-11.0.10. To measure times for Prusti, we timed the execution of running the `prusti-rustc` command line tool on each individual benchmark, setting the `check_overflows` flag to false. No additional configuration was made.

Table 1 summarizes statistics about the implementations, including lines of code (LOC). The LOC count has small differences between Flux and Prusti. This discrepancy is mostly due to differences in the way `RVec` has to be specified in Prusti, which sometimes requires adjustments to the code, as we explain in § 5.2.

5.1 Faster Verification

The columns **Time(s)** in table 1 show times taken by Flux and Prusti for each benchmark. While Flux requires only a couple of seconds to verify each benchmark, Prusti consistently takes an order of magnitude longer, taking more than 4 minutes to verify `fft`, the largest of the benchmarks. Note that Flux is faster despite having to spend time computing a fixpoint to synthesize loop invariant types, unlike Prusti, where this information is furnished by the user. We speculate that it may be because, with Prusti, the SMT solver has to *instantiate* and *check* quantified loop invariants which are known to cause performance issues in SMT solvers [Leino and Pit-Claudel 2016]. Nevertheless, further experimentation with Prusti may shed more light on the gap, and yield optimizations that can bring the verification times closer.

5.2 Compact API Specifications

The columns **Spec** in table 1 show the lines of code required for function specifications in Flux and Prusti. For the most part, the number of lines are similar, but slightly larger for Prusti, mostly due to the style of splitting annotations out into separate lines,*e.g.,* for pre- and post-conditions. However, in some important situations, Flux's type-based specifications allow for APIs that are shorter to write, faster to verify, and easier to *reuse*.

Quantifiers vs. Polymorphism In § 2.3 we showed a concise and precise interface for `RVec` which uses polymorphism to express quantified invariants over the elements of the vector. An interesting piece of this interface is `get_mut`, used to grant mutable access to the vector while maintaining the invariants over its elements. The simplest way to provide a comparable interface in Prusti is by defining a `store` function with the specification in fig. 9. (Prusti also supports the specification of `get_mut` using a more advance feature called *pledges*, but it has the same drawbacks as `store`.) This function takes a mutable reference to the vector, an index, and a value to store in that index. The specification `requires` the index to be within bounds and `ensures` (1) the vector has the same length after the function returns and (2) all the elements in the vector remain unchanged (3) except for the one being updated which gets the new value.

Prusti's quantified specification of `store` has two drawbacks. First, it makes verification slower: the signature uses a universally quantified formula, which makes it harder for the SMT solver to

```
// Rust Specification -----------------------------------------------------
fn store(&mut self, idx: usize, value: T)

// Prusti Specification ---------------------------------------------------
#[requires(idx < self.len())]
#[ensures(self.len() == old(self.len()))]
#[ensures(forall(|i: usize| (i < self.len() && i != idx) ==>
                                    self.lookup(i) == old(self.lookup(i))))]
#[ensures(self.lookup(idx) == value)]

// Flux Specification -----------------------------------------------------
fn store(self: &mut RVec<T, @n>, i: usize{i < n})
```

Fig. 9. The specifications for `RVec::store` in Rust, Prusti and Flux

discharge the verification conditions [Leino and Pit-Claudel 2016] created by clients of the library. Second, it prevents code reuse: the specification *must* reason about equality between the elements of the vector, meaning the vector can only store simple values for which equality is supported by the solver. Consequently, as equality between vectors is *not* supported, we cannot just simply use an `RVec<RVec<f32>>` to work over a collection of n-dimensional points as required by `kmeans`, but instead, the implementation in Prusti uses a trusted version of `RMat`: each n-dimensional point corresponds to a row in the matrix. Because individual rows cannot be accessed independently, we have to modify the code to pass around the entire matrix along with an index pointing to a particular row. The end result is that many of the critical properties cannot be verified and are hidden under the (trusted) implementation of `RMat`.

5.3 Fewer Annotations

The greatest payoff from refinement types is that by eschewing quantified assertions, they eliminate the annotation overhead for loop invariants. The column **Annot** in table 1 shows the number of lines taken by Prusti's *loop invariant* annotations. The annotation overhead for Prusti is non-trivial: about 14% of the implementation lines of code. In contrast, the column is missing for Flux as it automatically synthesizes the equivalent information via liquid typing § 4.

Easy Invariants via Typing For most of the benchmarks, loop invariants express either simple inequalities or tedious bookkeeping (*e.g.,* the length of a vector remains constant through a loop). While simple, they still have to be discovered and manually annotated by the user. The `fft` benchmark is a particularly egregious example, requiring a substantial amount of annotations, as it has a high number of (nested) loops that require annotation. The following snippet shows the annotations required for one of the loops:

```
body_invariant!(px.len() == n + 1 && py.len() == n + 1);
body_invariant!(i0 <= i1 && i1 <= i2 && i2 <= i3 && i3 <= n);
```

The first invariant asserts that the lengths of the vectors `px` and `py` stay constant through the loop. In Prusti, this must be spelled out as an invariant because the signature for `store` (fig. 9) says the output-length is the same as the input-length (`old`), forcing the verifier to explicitly propagate these equalities in the verification conditions. In contrast, as the reference is marked as `mut` (but not `strg`), Flux leaves the sizes unchanged and directly uses the *same* size-index during verification! The second specifies simple inequalities between `i0`, `i1`, `i2`, `i3` and `n`. As this is just a conjunction of quantifier free formulas, it is easily inferred by liquid typing.

Quantified Loop Invariants vs Polymorphism However, several benchmarks require complex universally quantified invariants in Prusti, but are equivalently handled by Flux's support for type polymorphism. For example, the function `kmp_table` from the `kmp` string matching benchmark takes as input a vector `p` of length `m` and computes a vector `t` of the same length containing indices into `p` (*i.e.,* integers between `0` and `m`). The function also uses two additional variables `i` and `j`, which are updated through the function's main loop. The following snippet shows the annotation required by Prusti to verify the implementation of `kmp_table`:

```
body_invariant!(forall(|x: usize| x < t.len() ==> t.lookup(x) < i));
body_invariant!(j < i && t.len() == p.len());
```

The first invariant is the critical one that asserts that in each iteration *every* element in `t` must be less than the current value of `i`. By using polymorphism to quantify over the elements of `t`, Flux can reduce the inference of this invariant to the inference of a quantifier free formula, liberating the user from manually annotating it.

6 RELATED WORK

Rust formal semantics RustBelt [Jung et al. 2017] provides a formalization of Rust aimed at proving that unsafe library implementations encapsulate their unsafe behavior under a well-typed interface. They achieve this by defining a semantic interpretation of Rust ownership types in Iris [Jung et al. 2018] and then proving that a library using unsafe code satisfies the predicates associated with the semantic interpretation of its interface. We base the formalization of Flux on RustBelt extending it with refinement types and use the same semantic argument to extend the system with libraries encapsulating unsafe code under a *refined* interface.

Weiss et al. [2019] follow a different approach at formalizing Rust. Their model Oxide formalizes a language which is closer to surface Rust. Oxide is based on a different interpretation of lifetimes as *provenance sets*, and resembles the prototype borrow checker implementation Polonius [Matsakis 2018]. It will be interesting to study how Flux fits in this new interpretation of lifetimes.

Refinement types and imperative code Refinement types were originally developed for the verification of functional programs [Freeman and Pfenning 1991; Rondon et al. 2008; Xi and Pfenning 1999b], but have also been used in the verification of heap-manipulating programs.

Based on earlier work on alias typing [Ahmed et al. 2007; Smith et al. 2000], Csolve [Rondon et al. 2010] extends C with liquid types to allow the verification of low-level programs using pointer arithmetic. Subsequent work extends Csolve to handle a restricted form of parallelism with shared state [Kawaguchi et al. 2012]. On a similar note, Alias Refinement Types (ART) [Bakst and Jhala 2016] builds on alias types to allow the verification of linked data structures. This line of work focuses on low-level manipulation of heap data through raw pointer using ad-hoc ways to control aliasing that have to be retrofitted into the language. In contrast, Flux builds on top of Rust references abstracting the spatial reasoning within Rust's type system. .

More recently, Sammler et al. [2021] proposed a type system that combines ownership and refinement types to provide automated verification for C programs. The focus is on providing a *foundational* tool that produces proofs in Coq and it follows an approach similar to RustBelt by defining a semantic interpretation of the type system in Iris. Our extension to RustBelt with refinement types closely resembles RefinedC, but their model of ownership is different from Rust references and requires the manual annotation of loops to track ownership.

Rust verification tools Several tools have been proposed for the automatic deductive verification of Rust programs; via encoding into Viper [Astrauskas et al. 2019; Müller et al. 2016]; generation of constrained Horn clauses [Bjørner et al. 2015; Matsushita et al. 2021]; and extraction into WhyML [Denis 2020; Filliâtre and Paskevich 2013]. All these tools leverage Rust's ownership types to abstract away the low level details of reasoning about aliasing, providing a specification language in a simple first-order program logic which is amenable for automatic verification. Using a program logic comes at the cost of complex user-specified universally quantified invariants. In contrast, by staying under a quantifier free fragment, our type-directed approach requires minimal programmer overhead.

Ullrich [2016] defined an encoding of safe Rust into a functional program, which can be interactively verified in Lean [Moura et al. 2015]. Similarly, Merigoux et al. [2021] define a translation into F*, but they target a fragment of Rust without mutation, which they use to verify cryptographic algorithms. Other tools have also focused on the *bounded* verification of Rust programs via model checking [Balasubramanian et al. 2017; VanHattum et al. 2022] or symbolic execution [Lindner et al. 2018].

7 CONCLUSIONS & FUTURE WORK

We presented Flux, which shows how logical refinements can be married with Rust's ownership mechanisms to yield ergonomic type-based verification for imperative code. Crucially, our design lets Flux express complex invariants by *composing* type constructors with simple quantifier-free logical predicates, and dually, let syntax directed subtyping to *decompose* complex reasoning about those invariants into efficiently decidable (quantifier free) validity queries over the predicates. This marriage makes verification ergonomic by allowing us to use predictable Horn-clause based machinery to automatically synthesize complicated loop-invariant annotations.

Of course, all marriages involve some compromise. By design, Flux restricts the specifications to those that can be expressed by the combination of type constructors and quantifier-free refinements. Program logic based methods like Prusti are more liberal. Their recursive heap predicates and universally quantified assertions permit specifications about the exact values in containers, and hence, verification of functional correctness properties which are currently out of Flux's reach. In future work, it would be interesting to see how to recoup such expressiveness, perhaps by incorporating techniques like abstract and bounded refinements, measures, and reflection [Vazou et al. 2015, 2013, 2018], that have proven effective in the purely functional setting.

REFERENCES

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^ 3: a linear language with locations. *Fundamenta Informaticae* 77, 4 (2007), 397–449.

Andrew W Appel. 2007. *Compiling with continuations.* Cambridge university press.

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

Alexander Bakst and Ranjit Jhala. 2016. Predicate abstraction for linked data structures. In *International Conference on Verification, Model Checking, and Abstract Interpretation.* Springer, 65–84.

Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems.* 156–161.

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II.* Springer, 24–51.

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58. https://doi.org/10.1007/978-3-642-36946-9_3

David G Clarke, John M Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 48–64.

R. L. Constable and S. F. Smith. 1987. Partial Objects In Constructive Type Theory. In *LICS*.

Benjamin Cosman and Ranjit Jhala. 2017. Local refinement typing. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–27.

Xavier Denis. 2020. *Deductive program verification for a language with a Rust-like typing discipline*. Ph.D. Dissertation. Université de Paris.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *European symposium on programming*. Springer, 125–128.

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 268–277.

Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Foundations and Trends® in Programming Languages* 6, 3–4 (2021), 159–317. https://doi.org/10.1561/2500000032

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. POPL (2017). https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. 2012. Deterministic parallelism via liquid effects. *ACM SIGPLAN Notices* 47, 6 (2012), 45–54.

K. Rustan M. Leino and Clément Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9779. Springer, 361–381. https://doi.org/10.1007/978-3-319-41528-4_20

Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No panic! Verification of Rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 108–114.

Niko Matsakis. 2018. An alias-based formulation of the borrow checker. https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-ali

Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-Based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 15 (oct 2021), 54 pages. https://doi.org/10.1145/3462205

Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. 2021. *Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust*. Ph.D. Dissertation. Inria.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2016. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*. Springer, 41–62.

James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings (Lecture Notes in Computer Science)*, Eric Jul (Ed.), Vol. 1445. Springer, 158–185. https://doi.org/10.1007/BFb0054091

Peter W O'hearn. 2004. Resources, concurrency and local reasoning. In *International Conference on Concurrency Theory*. Springer, 49–67.

John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

P. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI*.

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. *ACM Sigplan Notices* 45, 1 (2010), 131–144.

J. Rushby, S. Owre, and N. Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *TSE*.

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 158–174.

Frederick Smith, David Walker, and Greg Morrisett. 2000. Alias types. In *European Symposium on Programming*. Springer, 366–381.

Supplementary-Material. 2022. Liquid Rust: Supplementary Material.

John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context-and Flow-Sensitive Ownership Refinement Types for Imperative Programs.. In *ESOP*. 684–714.

Sebastian Ullrich. 2016. Simple verification of rust programs via functional purification. *Master's Thesis, Karlsruher Institut für Technologie (KIT)* (2016).

Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying Dynamic Trait Objects in Rust. (2022).

Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 48–61.

Niki Vazou, Patrick M Rondon, and Ranjit Jhala. 2013. Abstract refinement types. In *European Symposium on Programming*. Springer, 209–228.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.* 2, POPL (2018), 53:1–53:31. https://doi.org/10.1145/3158141

Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2019. Oxide: The essence of rust. *arXiv preprint arXiv:1903.00982* (2019).

H. Xi and F. Pfenning. 1999a. Dependent Types in Practical Programming. In *POPL*.

Hongwei Xi and Frank Pfenning. 1999b. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 214–227.

A SYNTAX

A.1 Surface syntax

The surface syntax of λ_{LR} mostly follows λ_{Rust}. As in λ_{Rust} the type system enforces programs to be in continuation-passing style to support control flow operators such as **return** and **break**.

Constants	c	$::=$	$\textbf{true} \mid \textbf{false}$	*booleans*
		$\mid$	$0, \pm 1, \ldots$	*integers*
Places	p	$:=$	x	*program variable*
		$\mid$	$*x$	*dereference*
R-values	rv	$::=$	c	*constants*
		$\mid$	$\&\textbf{mut}\ p$	*mutable (re)borrow*
		$\mid$	$\&p$	*shared (re)borrow*
Statements	S	$::=$	$p := rv$	*assignment*
		$\mid$	$p :=_n \textbf{copy}\ p$	*copy memory*
		$\mid$	$p :=_n \textbf{move}\ p$	*move memory*
Function body	F	$:=$	$\textbf{let}\ x = \textbf{new}(n)\ \textbf{in}\ F$	*let-bind*
		$\mid$	$\textbf{unpack}(x, v)\ \textbf{in}\ F$	*unpacking*
		$\mid$	$S; F$	*sequence*
		$\mid$	$\textbf{jump}\ k\langle\overline{r}\rangle(\overline{x})$	*jump to continuation*
		$\mid$	$\textbf{if}\ p\ \{F\}\ \textbf{else}\ \{F\}$	*if-then-else*
		$\mid$	$\textbf{call}\ f\langle\overline{r}\rangle(\overline{x})\ \textbf{ret}\ k$	*function call*
		$\mid$	$\textbf{letcont}\ k(\overline{x}) = F_1\ \textbf{in}\ F_2$	*continuation definition*
Function definition	D	$:=$	$\textbf{fn}\ f(\overline{x})\ \textbf{ret}\ k\ \{F\}$	

A.2 Translation into λ_{Rust}

The surface syntax is given operational behavior by defining them as derived forms in the core calculus of RustBelt.

$$
\begin{aligned}
\textbf{let}\ x = e_1\ \textbf{in}\ e_2 &:= (\textbf{rec}\ \_([x])\ :=\ e_2)(e_1)\\
\textbf{fn}\ f(\overline{x})\ \textbf{ret}\ k\ \{F\} &:= \textbf{let}\ f = \textbf{alloc}(1)\ \textbf{in}\ f :=_{\mathsf{na}} (\textbf{rec}\ \_([k] +\!\!+ \overline{x})\ :=\ F)\\
e_1; e_2 &:= \textbf{let}\ \_ = e_1\ \textbf{in}\ e_2\\
\textbf{letcont}\ k(\overline{x})\ =\ e_1\ \textbf{in}\ e_2 &:= \textbf{let}\ k = (\textbf{rec}\ k(\overline{x})\ :=\ e_1)\ \textbf{in}\ e_2\\
\textbf{jump}\ k\langle\_\rangle(\overline{e}) &:= k(\overline{e})\\
\textbf{call}\ f\langle\_\rangle(\overline{e})\ \textbf{ret}\ k &:= ({}^{*\mathsf{na}} f)([k] +\!\!+ \overline{e})\\
\textbf{false} &:= 0\\
\textbf{true} &:= 1\\
\textbf{if}\ e_0\ \{e_1\}\ \textbf{else}\ \{e_2\} &:= \textbf{case}\ e_0\ \textbf{of}\ [e_1, e_2]\\
e &:= {}^{\mathsf{na}} e\\
\&\textbf{mut}\ e &:= e\\
\&e &:= e\\
\textbf{new} &:= \textbf{rec}\ \mathit{new}(n)\ :=\\
&\qquad \textbf{if}\ n == 0\ \textbf{then}\ (42, 1337)\ \textbf{else}\ \textbf{alloc}(n)\\
\textbf{delete} &:= \textbf{rec}\ \mathit{delete}(n, \mathit{ptr})\ :=\\
&\qquad \textbf{if}\ n == 0\ \textbf{then}\ ☣\ \textbf{else}\ \textbf{free}(n, \mathit{ptr})\\
\textbf{memcpy} &:= \textbf{rec}\ \mathit{memcpy}(\mathit{dst}, n, \mathit{src})\ :=\\
&\qquad \textbf{if}\ n \le 0\ \textbf{then}\ ☣\ \textbf{else}\\
&\qquad \mathit{dst}.0 := \mathit{src}.0;\\
&\qquad \mathit{memcpy}(\mathit{dst}.1, n - 1, \mathit{src}.1)\\
e_1 := e_2 &:= e_1 :=_{\mathsf{na}} e_2\\
e_1 :=_n \textbf{copy}\ e_2 &:= \textbf{memcopy}(e_1, n, e_2)\\
e_1 :=_n \textbf{move}\ e_2 &:= \textbf{memcopy}(e_1, n, e_2)\\
\textbf{skip} &:= \textbf{let}\ \_ = ☣\ \textbf{in}\ ☣\\
\textbf{newlft} &:= ☣\\
\textbf{enflft} &:= \textbf{skip}\\
\textbf{unpack}(x, v)\ \textbf{in}\ F &:= F
\end{aligned}
$$

A.3 Types and contexts

Types	τ	$:=$	$B\langle r\rangle$	*index type, e.g.,* $\textbf{int}\langle 1\rangle$
		$\mid$	$\{v.\ B\langle v\rangle \mid r\}$	*existential type, e.g.,* $\{v.\ \textbf{int}\langle v\rangle \mid v > 0\}$
		$\mid$	$\textbf{ptr}(\ell)$	*pointer to location* ℓ
		$\mid$	$\&_\mu\ \tau$	*borrowed reference (mutable or shared)*
		$\mid$	$\lightning_n$	*uninitialized memory of size* n
Locations	ℓ	$:=$	x	*program variable*
		$\mid$	ρ	*location ghost variable*
Base Types	B	$:=$	**int**	*integers*
		$\mid$	**bool**	*booleans*
Modifier	μ	$:=$	**mut**	*mutable reference*
		$\mid$	**shr**	*shared reference*

$$
\begin{array}{rrcl}
\textit{Function signatures} & s & := & \forall \overline{v : \sigma}.\ \mathbf{fn}(r; \overline{x}.\ \mathbf{T}) \to \rho_{\mathsf{o}}.\ \mathbf{T} \\
\textit{Refinements} & r & := & v \mid \kappa \mid \ell \mid \mathsf{true} \mid \mathsf{false} \mid r + r \mid r > r \mid r = r \mid \ldots \\
\textit{Sorts} & \sigma & := & \mathbf{int} \mid \mathbf{bool} \mid \mathbf{loc} \\
\textit{Variable contexts} & \mathbf{\Delta} & := & \emptyset \mid \mathbf{\Delta}, v : \sigma \mid \mathbf{\Delta}, r \\
\textit{Continuation contexts} & \mathbf{K} & := & \emptyset \mid \mathbf{K}, k : \forall \overline{v : \sigma}.\ \mathbf{cont}(r; \overline{x}.\ \mathbf{T}) \\
\textit{Location contexts} & \mathbf{T} & := & \emptyset \mid \mathbf{T}, \ell \mapsto \tau \\
\textit{Global environements} & \mathbf{\Sigma} & := & \emptyset \mid \mathbf{\Sigma}, f : s
\end{array}
$$

B TYPE SYSTEM

B.1 Well-formedness

The well-formedness judgments ($\vdash_{\mathsf{wf}}$) check the binding structure of the grammar and well-sortedness of expressions. We only show a sample of well-worted expressions, in the implementation these are limited by what can be handled by smt solvers.

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

$$\mathbf{\Delta} \vdash_{\mathsf{wf}} \lightning_n \qquad \frac{\mathbf{\Delta} \vdash r : \mathbf{bool}}{\mathbf{\Delta} \vdash_{\mathsf{wf}} \mathbf{bool}\langle r \rangle} \qquad \frac{\mathbf{\Delta} \vdash r : \mathbf{int}}{\mathbf{\Delta} \vdash_{\mathsf{wf}} \mathbf{int}\langle r \rangle}$$

$$\frac{\sigma = \mathsf{sort}(B) \qquad \mathbf{\Delta}, v : \sigma \vdash r : \mathbf{bool} \qquad \mathbf{\Delta}, v : \sigma \vdash_{\mathsf{wf}} B\langle v \rangle}{\mathbf{\Delta} \vdash_{\mathsf{wf}} \{v.\ B\langle v \rangle \mid r\}} \qquad \frac{\ell : \mathbf{loc} \in \mathbf{\Delta}}{\mathbf{\Delta} \vdash_{\mathsf{wf}} \mathbf{ptr}(\ell)} \qquad \frac{\mathbf{\Delta} \vdash \kappa : \mathbf{lft} \qquad \mathbf{\Delta} \vdash_{\mathsf{wf}} \tau}{\mathbf{\Delta} \vdash_{\mathsf{wf}} \&_{\mu}\ \tau}$$

Well-formed location contexts $\boxed{\mathbf{\Delta} \vdash_{\mathsf{wf}} \mathbf{T}}$

$$\mathbf{\Delta} \vdash_{\mathsf{wf}} \emptyset \qquad \frac{\mathbf{\Delta} \vdash_{\mathsf{wf}} \mathbf{T} \qquad \mathbf{\Delta} \vdash \ell : \mathbf{loc} \qquad \mathbf{\Delta} \vdash_{\mathsf{wf}} \tau}{\mathbf{\Delta} \vdash_{\mathsf{wf}} \mathbf{T}, \ell \mapsto \tau}$$

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

$$\mathbf{\Delta} \vdash_{\mathsf{wf}} \emptyset \qquad \frac{\mathbf{\Delta} \vdash_{\mathsf{wf}} \mathbf{K} \qquad \mathbf{\Delta}, \overline{v : \sigma}, \overline{x} : \mathbf{loc} \vdash_{\mathsf{wf}} \mathbf{T}}{\mathbf{\Delta} \vdash_{\mathsf{wf}} \mathbf{K}, k : \forall \overline{v : \sigma}.\ \mathbf{cont}(\mathbf{L}; \mathbf{T}.)}$$

Well-sorted expressions $\boxed{\mathbf{\Delta} \vdash r : \sigma}$

$$\frac{v : \sigma \in \mathbf{\Delta}}{\mathbf{\Delta} \vdash v : \sigma} \qquad \mathbf{\Delta} \vdash n : \mathbf{int} \qquad \mathbf{\Delta} \vdash \mathsf{true} : \mathbf{bool} \qquad \mathbf{\Delta} \vdash \mathsf{false} : \mathbf{bool}$$

$$\frac{\mathbf{\Delta} \vdash r_1 : \mathbf{int} \qquad \mathbf{\Delta} \vdash r_2 : \mathbf{int}}{\mathbf{\Delta} \vdash r_1 + r_2 : \mathbf{int}} \qquad \frac{\mathbf{\Delta} \vdash r_1 : \mathbf{int} \qquad \mathbf{\Delta} \vdash r_2 : \mathbf{int}}{\mathbf{\Delta} \vdash r_1 > r_2 : \mathbf{bool}}$$

B.2 Auxiliary definitions

B.2.1 Size. The *size* of a type says how many memory locations a type spans. It is defined as follows:

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{size}(\mathbf{bool}\langle r\rangle) & := & 1 & \mathtt{size}(\&_\mu\, \tau) & := & 1 \\ \mathtt{size}(\mathbf{int}\langle r\rangle) & := & 1 & \mathtt{size}(\mathbf{ptr}(\ell)) & := & 1 \\ \mathtt{size}(\lightning_n) & := & n & \mathtt{size}(\{v.\, B\langle v\rangle \mid r\}) & := & \mathtt{size}(B\langle v\rangle) \end{array}$$

B.2.2 Copy. Some types are *copy* which means they can be used arbitrarily often. This is expressed by the following judgment.

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

$$\mathbf{bool}\langle r\rangle\ \mathtt{copy} \qquad \mathbf{int}\langle r\rangle\ \mathtt{copy} \qquad \&_{\mathbf{shr}}\, \tau\ \mathtt{copy}$$

B.2.3 Reference modifiers order. We define the order of reference modifiers ($\preceq$) as the reflexive closure of $\mathbf{shr} \preceq \mathbf{mut}$.

B.2.4 Sorts. Each base type is associated with a *sort*.

$$\mathtt{sort}(\mathbf{int}) := \mathbf{int}$$
$$\mathtt{sort}(\mathbf{bool}) := \mathbf{bool}$$

B.3 Lifetime context judgments

The following judgments express various properties of lifetime contexts.

Lifetime inclusion $\boxed{\mathbf{E}; \mathbf{L} \vdash \kappa \sqsubseteq \kappa'}$

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

$$\frac{\mathbf{E}; \mathbf{L} \vdash \kappa_1 \sqsubseteq \kappa_2 \qquad \mathbf{E}; \mathbf{L} \vdash \kappa_2 \sqsubseteq \kappa_3}{\mathbf{E}; \mathbf{L} \vdash \kappa_1 \sqsubseteq \kappa_3}$$

Lifetime liveness $\boxed{\mathbf{E};\mathbf{L} \vdash \kappa \text{ alive}}$

$$\mathbf{E};\mathbf{L} \vdash \textbf{static} \text{ alive} \qquad \frac{\kappa \sqsubseteq_{\mathrm{l}} \overline{\kappa} \in \mathbf{L} \qquad \forall i \in \overline{\kappa}.\ \mathbf{E};\mathbf{L} \vdash \kappa_i \text{ alive}}{\mathbf{E};\mathbf{L} \vdash \kappa \text{ alive}}$$

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

Local lifetime context inclusion $\boxed{\Delta \vdash \mathbf{L} \Rightarrow \mathbf{L}'}$

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

External lifetime context satisfiability $\boxed{\mathbf{\Delta} \mid \mathbf{E};\mathbf{L} \vdash \mathbf{E}'}$

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

B.4 Subtyping and context inclusion

B.4.1 Entailment. The entailment judgment $[\![\mathbf{\Delta} \vdash r]\!]$ checks that r holds true under context $\mathbf{\Delta}$. The exact definition depends on the underlying logic. The implementation discharge these judgments with an SMT solver.

B.4.2 Subtyping. The main subtyping supported by rust is lifetime inclusion. We extend it with subtyping between refinement types. Additionally, type constructors have structural rules witnessing their variance.

Subtyping $\boxed{\mathbf{\Delta} \vdash \tau_1 \preccurlyeq \tau_2}$

S-Refl
$$\mathbf{\Delta} \vdash \tau \preccurlyeq \tau$$

S-Trans
$$\frac{\mathbf{\Delta} \vdash \tau_1 \preccurlyeq \tau_2 \qquad \mathbf{\Delta} \vdash \tau_2 \preccurlyeq \tau_3}{\mathbf{\Delta} \vdash \tau_1 \preccurlyeq \tau_3}$$

S-RType
$$\frac{[\![\mathbf{\Delta} \vdash r_1 = r_2]\!]}{\mathbf{\Delta} \vdash B\langle r_1 \rangle \preccurlyeq B\langle r_2 \rangle}$$

S-Unpack
$$\frac{\sigma = \mathtt{sort}(B) \qquad \mathbf{\Delta}, v : \sigma, r \vdash B\langle v \rangle \preccurlyeq \tau}{\mathbf{\Delta} \vdash \{v.\ B\langle v \rangle \mid r\} \preccurlyeq \tau}$$

S-Exists
$$\frac{[\![\mathbf{\Delta} \vdash r_2[r_1/v]]\!]}{\mathbf{\Delta} \vdash B\langle r_1 \rangle \preccurlyeq \{v.\ B\langle v \rangle \mid r_2\}}$$

S-Bor-Shr
$$\frac{\mathbf{\Delta} \vdash \tau_1 \preccurlyeq \tau_2}{\mathbf{\Delta} \vdash \&_{\mathbf{shr}}\ \tau_1 \preccurlyeq \&_{\mathbf{shr}}\ \tau_2}$$

S-Bor-Mut
$$\frac{\mathbf{\Delta} \vdash \tau_1 \preccurlyeq \tau_2 \qquad \mathbf{\Delta} \vdash \tau_2 \preccurlyeq \tau_1}{\mathbf{\Delta} \vdash \&_{\mathbf{mut}}\ \tau_1 \preccurlyeq \&_{\mathbf{mut}}\ \tau_2}$$

B.4.3 Location context inclusion. The following judgment states when it is safe to jump from context $\mathbf{T}_1$ to $\mathbf{T}_2$. The are structure rules for *weakening*, *permutation* and *framing*. The rule C-Sub allows applying subtyping. The rule C-Bor allows coercing a pointer into a borrowed reference. We do not need a rule for reborrowing as in λ_{Rust} as reborrowing is handled explicitly when checking r-values.

Location context inclusion $\boxed{\boldsymbol{\Delta} \vdash \mathbf{T}_1 \Rightarrow \mathbf{T}_2}$

$$\text{C-Perm}\ \frac{\mathbf{T}' \text{ is a permutation of } \mathbf{T}}{\boldsymbol{\Delta} \vdash \mathbf{T} \Rightarrow \mathbf{T}'} \qquad \text{C-Weaken}\ \boldsymbol{\Delta} \vdash \mathbf{T}, \mathbf{T}' \Rightarrow \mathbf{T} \qquad \text{C-Frame}\ \frac{\boldsymbol{\Delta} \vdash \mathbf{T}_1 \Rightarrow \mathbf{T}_2}{\boldsymbol{\Delta} \vdash \mathbf{T}', \mathbf{T}_1 \Rightarrow \mathbf{T}', \mathbf{T}_2}$$

$$\text{C-Sub}\ \frac{\boldsymbol{\Delta} \vdash \tau_1 \preccurlyeq \tau_2}{\boldsymbol{\Delta} \vdash \ell_1 \mapsto \tau_1 \Rightarrow \ell_2 \mapsto \tau_2} \qquad \text{C-Bor}\ \boldsymbol{\Delta} \vdash \ell_1 \mapsto \tau, \ell_2 \mapsto \mathbf{ptr}(\ell_1) \Rightarrow \ell_1 \overset{\dagger\kappa}{\longmapsto} \tau, \ell_2 \mapsto \&_\mu \tau$$

$$\text{C-Block}\ \boldsymbol{\Delta} \vdash \ell_1 \mapsto \tau \Rightarrow \ell_1 \overset{\dagger\kappa}{\longmapsto} \tau$$

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

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

$$\frac{\Delta \mid \mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}' \qquad \Delta, \overline{v : \sigma}, \overline{x} : \mathbf{loc} \vdash \mathbf{T} \Rightarrow \mathbf{T}'}{\Delta \mid \mathbf{E} \vdash \mathbf{K}, k : \forall \overline{v : \sigma}.\, \mathbf{cont}(\mathbf{L}; \mathbf{T}.\ \Rightarrow) \mathbf{K}', k : \forall \overline{v : \sigma}.\, \mathbf{cont}(\mathbf{L}; \mathbf{T}'.)}$$

B.4.4 Type unblocking. The following judgment expresses that when κ ends, then we can unblock" the parts of the location context that is blocked by κ.

Location context unblocking $\boxed{\mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}$

$$\emptyset \Rightarrow^{\dagger\kappa} \emptyset \qquad \frac{\mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}{\mathbf{T}_1, \ell \mapsto \tau \Rightarrow^{\dagger\kappa} \mathbf{T}_2, \ell \mapsto \tau} \qquad \frac{\mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}{\mathbf{T}_1, \ell \overset{\dagger\kappa}{\longmapsto} \tau \Rightarrow^{\dagger\kappa} \mathbf{T}_2, \ell \mapsto \tau}$$

$$\frac{\mathbf{T}_1 \Rightarrow^{\dagger\kappa} \mathbf{T}_2}{\mathbf{T}_1, \ell \overset{\dagger\kappa'}{\longmapsto} \tau \Rightarrow^{\dagger\kappa} \mathbf{T}_2, \ell \overset{\dagger\kappa'}{\longmapsto} \tau}$$

B.5 Well-typed programs

Well-typed Programs $\boxed{\boldsymbol{\Sigma} \vdash P}$

$$\frac{\forall D_i \in \overline{D}.\, \boldsymbol{\Sigma} \vdash D_i}{\boldsymbol{\Sigma} \vdash \overline{D}}\ \text{T-Pg}$$

Well-typed Function definitions $\boxed{\Sigma \vdash D}$

$$\frac{\forall D_i \in \overline{D}.\, \Sigma \vdash D_i}{\Sigma \vdash \overline{D}}\ \text{T-Pg} \qquad \frac{\begin{array}{c}\Sigma(f) = \forall \overline{v:\sigma}.\, \mathbf{fn}(r; \overline{x}.\, \mathbf{T}_i) \to \rho_o.\, \mathbf{T}_o \\ \Sigma; \overline{x} : \mathbf{loc}, \overline{v:\sigma}, r \mid \mathbf{T}_i; k : \mathbf{cont}(\mathbf{true}; \rho_o.\, \mathbf{T}_o) \vdash F\end{array}}{\Sigma \vdash \mathbf{fn}\ f(\overline{x})\ \mathbf{ret}\ k\ \{F\}}\ \text{T-Def}$$

Well-typed function definitions $\boxed{\Sigma \vdash D}$

$$\text{T-def}\ \frac{\begin{array}{c}\Sigma(f) = \forall \overline{v:\sigma}.\, \mathbf{fn}(r; \overline{x}.\, \mathbf{T}_i) \to \rho_o.\, \mathbf{T}_o \\ \overline{x} : \mathbf{loc}, \overline{v:\sigma}, r \mid \mathbf{T}_i; k : \mathbf{cont}(\mathfrak{f} \sqsubseteq_{\mathrm{l}} []; \mathbf{T}_o.\, \vdash) F\end{array}}{\Sigma \vdash \mathbf{fn}\ f(\overline{x})\ \mathbf{ret}\ k\ \{F\}}$$

Well-typed function bodies $\boxed{\Sigma; \Delta \mid \mathbf{T}; \mathbf{K} \vdash F}$

$$\text{T-consequence}\ \frac{\Delta \vdash \mathbf{L} \Rightarrow \mathbf{L}' \qquad \Delta \vdash \mathbf{T} \Rightarrow \mathbf{T}' \qquad \Delta \mid \mathbf{E} \vdash \mathbf{K} \Rightarrow \mathbf{K}' \qquad \Delta \mid \mathbf{K}'; \mathbf{T}' \vdash F}{\Delta \mid \mathbf{K}; \mathbf{T} \vdash F}$$

$$\text{T-Let}\ \frac{\Delta, x : \mathbf{loc} \mid \mathbf{T}, x \mapsto \lightning_n; \mathbf{K} \vdash F}{\Delta \mid \mathbf{K}; \mathbf{T} \vdash \mathbf{let}\ x = \mathbf{new}(n)\ \mathbf{in}\ F} \qquad \text{T-unpack}\ \frac{\sigma = \mathtt{sort}(B) \qquad \Delta, v : \sigma, r[v/b] \mid \mathbf{K}; \mathbf{T}, x \mapsto B\langle v\rangle \vdash F}{\Delta \mid \mathbf{K}; \mathbf{T}, x \mapsto \{b.\, B\langle b\rangle \mid r\} \vdash \mathbf{unpack}(x, v)\ \mathbf{in}\ F}$$

$$\text{T-Seq}\ \frac{\Delta_1 \mid \mathbf{T}_1 \vdash S \dashv \Delta_2 \mathbf{T}_2 \qquad \Delta_1, \Delta_2 \mid \mathbf{K}; \mathbf{T}_2, \mathbf{T} \vdash F}{\Delta_1 \mid \mathbf{K}; \mathbf{T}_1, \mathbf{T} \vdash S; F} \qquad \text{T-Jump}\ \frac{\begin{array}{c}\mathbf{K}(k) = \forall \overline{v:\sigma}.\, \mathbf{cont}(r'; \overline{y}.\, \mathbf{T}') \qquad \overline{\Delta \vdash r : \sigma} \\ \theta = [\overline{x}/\overline{y}][\overline{r}/\overline{v}] \qquad \Delta \vdash \mathbf{T} \Rightarrow \theta \cdot \mathbf{T}' \qquad [\![\Delta \vdash \theta \cdot r']\!]\end{array}}{\Delta \mid \mathbf{K}; \mathbf{T} \vdash \mathbf{jump}\ k\langle\overline{r}\rangle(\overline{x})}$$

$$\text{T-If}\ \frac{\begin{array}{c}\mathbf{T} \vdash p \leftrightarrows \mathbf{bool}\langle r\rangle \\ \Delta, r \mid \mathbf{K}; \mathbf{T} \vdash F_1 \qquad \Delta, \neg r \mid \mathbf{K}; \mathbf{T} \vdash F_2\end{array}}{\Delta \mid \mathbf{K}; \mathbf{T} \vdash \mathbf{if}\ p\ \{F_1\}\ \mathbf{else}\ \{F_2\}}$$

$$\text{T-Cont}\ \frac{\begin{array}{c}\Delta, \overline{v:\sigma}, \overline{x} : \mathbf{loc} \mid \mathbf{K}, k : \forall \overline{v:\sigma}.\, \mathbf{cont}(\mathbf{L}_1; \mathbf{T}'.;)\mathbf{T}' \vdash F_1 \\ \Delta \mid \mathbf{K}, k : \forall \overline{v:\sigma}.\, \mathbf{cont}(\mathbf{L}_1; \mathbf{T}'.;)\mathbf{T} \vdash F_2\end{array}}{\Delta \mid \mathbf{K}; \mathbf{T} \vdash \mathbf{letcont}\ k(\overline{x})\ =\ F_1\ \mathbf{in}\ F_2}$$

$$\text{T-call}\ \frac{\begin{array}{c}\mathbf{K}(k) = \mathbf{cont}(\mathbf{true}; z.\, \mathbf{T}_k) \qquad \Sigma(f) = \forall \overline{v:\sigma}.\, \mathbf{fn}(r; \overline{y}.\, \mathbf{T}_i) \to \rho_{\mathrm{o}}.\, \mathbf{T}_o \\ \theta = [\overline{r}/\overline{v}][\overline{x}/\overline{y}] \qquad \overline{\Delta \vdash r : \sigma} \qquad [\![\Delta \vdash \theta \cdot r]\!] \\ \Delta \vdash \mathbf{T}_1 \Rightarrow \theta \cdot \mathbf{T}_i \qquad \Delta \vdash \mathbf{T}_2, \theta \cdot \mathbf{T}_o \Rightarrow \mathbf{T}_k[\rho_{\mathrm{o}}/z]\end{array}}{\Sigma; \Delta \mid \mathbf{K}; \mathbf{T}_1, \mathbf{T}_2 \vdash \mathbf{call}\ f\langle\overline{r}\rangle(\overline{x})\ \mathbf{ret}\ k}$$

Well-typed statements $\boxed{\boldsymbol{\Delta}_i \mid \mathbf{T}_i \vdash S \dashv \mathbf{T}_o}$

$$\textsc{T-assign}\ \frac{\boldsymbol{\Delta} \mid \mathbf{T}_i \vdash rv \rightsquigarrow \tau \dashv \mathbf{T}' \quad \boldsymbol{\Delta} \mid \mathbf{T}' \vdash p \leftsquigarrow \tau \dashv \mathbf{T}_o}{\boldsymbol{\Delta} \mid \mathbf{T}_i \vdash p := rv \dashv \mathbf{T}_o} \qquad \textsc{T-copy}\ \frac{\mathbf{T}_1 \vdash p \overset{\text{cp}}{\rightsquigarrow} \tau \qquad \mathtt{size}(\tau) = n \quad \boldsymbol{\Delta} \mid \mathbf{T}_1 \vdash p \leftsquigarrow \tau \dashv \mathbf{T}_2}{\boldsymbol{\Delta} \mid \mathbf{T}_1 \vdash p_1 :=_n \mathbf{copy}\ p_2 \dashv \mathbf{T}_o}$$

$$\textsc{T-move}\ \frac{\mathbf{T}_1 \vdash p \overset{\text{mv}}{\rightsquigarrow} \tau \dashv \mathbf{T}_2 \qquad \mathtt{size}(\tau) = n \quad \boldsymbol{\Delta} \mid \mathbf{T}_2 \vdash p \leftsquigarrow \tau \dashv \mathbf{T}_3}{\boldsymbol{\Delta} \mid \mathbf{T}_1 \vdash p_1 :=_n \mathbf{move}\ p_2 \dashv \mathbf{T}_3}$$

Well-typed r-values $\boxed{\boldsymbol{\Delta} \mid \mathbf{T}_i \vdash rv \rightsquigarrow \tau \dashv \mathbf{T}_o}$

$$\textsc{T-int}\ \boldsymbol{\Delta} \mid \mathbf{T} \vdash n \rightsquigarrow \mathbf{int}\langle n \rangle \dashv \mathbf{T} \qquad \textsc{T-true}\ \boldsymbol{\Delta} \mid \mathbf{T} \vdash \mathbf{true} \rightsquigarrow \mathbf{bool}\langle \mathsf{true} \rangle \dashv \mathbf{T}$$

$$\textsc{T-false}\ \boldsymbol{\Delta} \mid \mathbf{T} \vdash \mathbf{false} \rightsquigarrow \mathbf{bool}\langle \mathsf{false} \rangle \dashv \mathbf{T} \qquad \textsc{T-bor-shr}\ \frac{\mathbf{T}_i \vdash p \overset{\&\mathbf{shr}}{\rightsquigarrow} \tau \dashv \mathbf{T}_o}{\boldsymbol{\Delta} \mid \mathbf{T}_i \vdash \&p \rightsquigarrow \tau \dashv \mathbf{T}_o}$$

$$\textsc{T-bor-mut}\ \frac{\mathbf{T}_i \vdash p \overset{\&\mathbf{mut}}{\rightsquigarrow} \tau \dashv \mathbf{T}_o}{\boldsymbol{\Delta} \mid \mathbf{T}_i \vdash \&\mathbf{mut}\ p \rightsquigarrow \tau \dashv \mathbf{T}_o}$$

Type copying $\boxed{\mathbf{T} \vdash p \overset{\text{cp}}{\rightsquigarrow} \tau}$

$$\textsc{T-copy-own}\ \frac{\tau\ \mathtt{copy}}{\ell \mapsto \tau \vdash \ell \overset{\text{cp}}{\rightsquigarrow} \tau} \qquad \textsc{T-copy-bor}\ \frac{\tau\ \mathtt{copy}}{\ell \mapsto \&_\mu\, \tau \vdash *\ell \overset{\text{cp}}{\rightsquigarrow} \tau} \qquad \textsc{T-copy-ptr}\ \frac{\tau\ \mathtt{copy}}{\ell_1 \mapsto \tau, \ell_2 \mapsto \mathbf{ptr}(\ell_1) \vdash *\ell_2 \overset{\text{cp}}{\rightsquigarrow} \tau}$$

Type moving $\boxed{\mathbf{T}_i \vdash p \overset{\text{mv}}{\rightsquigarrow} \tau \dashv \mathbf{T}_o}$

$$\textsc{T-move-own}\ \frac{n = \mathtt{size}(\tau)}{x \mapsto \tau \vdash x \overset{\text{mv}}{\rightsquigarrow} \tau \dashv x \mapsto \lightning_n} \qquad \textsc{T-move-ptr}\ \frac{n = \mathtt{size}(\tau)}{x \mapsto \mathbf{ptr}(\ell), \ell \mapsto \tau \vdash *\ell \overset{\text{mv}}{\rightsquigarrow} \tau \dashv x \mapsto \mathbf{ptr}(\ell), \ell \mapsto \lightning_n}$$

Type writing

$$\boxed{\boldsymbol{\Delta} \mid \mathbf{T}_i \vdash p \leftsquigarrow \tau \dashv \mathbf{T}_o}$$

T-WRITE-OWN
$$\frac{\mathtt{size}(\tau) = \mathtt{size}(\tau')}{\boldsymbol{\Delta} \mid x \mapsto \tau' \vdash x \leftsquigarrow \tau \dashv x \mapsto \tau}$$

T-WRITE-PTR
$$\frac{\mathtt{size}(\tau) = \mathtt{size}(\tau')}{\boldsymbol{\Delta} \mid x \mapsto \mathbf{ptr}(\ell), \ell \mapsto \tau' \vdash x \leftsquigarrow \tau \dashv x \mapsto \mathbf{ptr}(\ell), \ell \mapsto \tau}$$

T-WRITE-BOR
$$\frac{\boldsymbol{\Delta} \vdash \tau \preccurlyeq \tau'}{\boldsymbol{\Delta} \mid \ell \mapsto \&_{\mathbf{mut}}\, \tau' \vdash *x \leftsquigarrow \tau \dashv \ell \mapsto \&_{\mathbf{mut}}\, \tau'}$$

Type borrowing

$$\boxed{\mathbf{T}_i \vdash p \overset{\&\mu}{\rightsquigarrow} \tau \dashv \mathbf{T}_o}$$

T-BOR-OWN
$$\frac{x \mapsto \tau \in \mathbf{T}}{\mathbf{T} \vdash x \overset{\&\mu}{\rightsquigarrow} \mathbf{ptr}(\ell) \dashv \mathbf{T}}$$

T-BOR-PTR
$$\frac{x \mapsto \mathbf{ptr}(\ell) \in \mathbf{T}}{\mathbf{T} \vdash *x \overset{\&\mu}{\rightsquigarrow} \mathbf{ptr}(\ell) \dashv \mathbf{T}}$$

T-REBOR
$$\frac{\mu \preceq \mu'}{x \mapsto \&_{\mu'}\, \tau \vdash *x \overset{\&\mu}{\rightsquigarrow} \&_{\mu}\, \tau \dashv x \overset{\dagger\kappa}{\longmapsto} \&_{\mu'}\, \tau}$$

C λ_{LR} MODEL

C.1 Types

C.1.1 Semantic Definitions. Types are interpreted in the model of RustBelt. We write down the definition of a type here for convenience.

Definition C.1 (Semantic Type). A *semantic type* is a tuple $(\text{size} \in \mathbb{N}, \text{own} \in \mathit{TId} \times \mathit{list}(\mathit{Val}) \to \mathit{iProp}, \text{shr} \in \mathit{Lft} \times \mathit{TId} \times \mathit{Loc} \to \mathit{iProp})$ such that

$$\forall t, \overline{v}.\, \text{own}(t, \overline{v}) \Rightarrow |\overline{v}| = \text{size} \qquad \text{(Ty-size)}$$
$$\forall \kappa, t, \ell.\, \text{shr}(\kappa, t, \ell) \Rightarrow \Box\, \text{shr}(\kappa, t, \ell) \qquad \text{(Ty-shr-persistent)}$$
$$\forall \kappa, t, \ell.\, \&^{\kappa}_{\mathbf{full}}(\ell \mapsto \text{own}(t)) * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\text{lft}}} \text{shr}(\kappa, t, \ell) * [\kappa]_q \qquad \text{(Ty-share)}$$
$$\forall \kappa, \kappa', t, \ell.\, \kappa' \sqsubseteq \kappa \Rightarrow \text{shr}(\kappa, t, \ell) \Rightarrow \text{shr}(\kappa', t, \ell) \qquad \text{(Ty-shr-mono)}$$

We also define the semantic of base types which are the basis to form indexed types. As with types they have a size, an ownership, and a sharing predicates. The ownership and share predicates are extended with an extra value corresponding to the index. The size predicate *cannot* depend on the index. Additionally, base types are associated to a sort.

Definition C.2 (Semantic Base Type). A *semantic base type* is a tuple $(\text{sort} \in \mathit{Type}, \text{size} \in \mathbb{N}, \text{own} \in \text{sort} \times \mathit{TId} \times \mathit{list}(\mathit{Val}) \to \mathit{iProp}, \text{shr} \in \text{sort} \times \mathit{Lft} \times \mathit{TId} \times \mathit{Loc} \to \mathit{iProp})$ such that for all $v \in \text{sort}$ $(\text{size}, \text{own}(v), \text{shr}(v))$ is a type.

Definition C.3 (Semantic subtyping).

$$\tau_1 \sqsubseteq^{\text{ty}} \tau_2 := \tau_1.\texttt{size} = \tau_2.\texttt{size} \wedge (\Box\, \forall t, \overline{v}.\, \tau_1.\texttt{own}(t, \overline{v}) \Rightarrow \tau_2.\texttt{own}(t, \overline{v})) \\ \wedge (\Box\, \forall \kappa, t, \ell.\, \tau_1.\texttt{shr}(\kappa, t, \ell) \Rightarrow \tau_2.\texttt{shr}(\kappa, t, \ell))$$

C.1.2 Auxiliary definitions.

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

C.1.3 Modeling base types.

$$[\![\mathbf{int}]\!] := \text{Singleton}(\mathbb{Z})$$
$$[\![\mathbf{bool}]\!] := \text{Singleton}(\mathbb{B})$$

$$
\begin{aligned}
[\![\mathbf{vec}\langle\tau\rangle]\!] := \text{TypeCon}\,\{\ & \text{sort} := \mathbb{Z} \\
; \ & \text{size} := 3 \\
; \ & \text{own} := \lambda l, t, \overline{v}. \\
& \qquad \overline{v} = [l, c, p] * l \le c * \\
& \qquad \textstyle\ast_{i=0}^{l}(\exists \overline{w}.\, p + i \cdot [\![\tau]\!]_\gamma.\mathsf{size} \mapsto \overline{w} * \triangleright [\![\tau]\!]_\gamma.\mathsf{own}(t, \overline{w})) * \\
& \qquad \textstyle\ast_{i=l\cdot [\![\tau]\!]_\gamma.\mathsf{size}}^{c\cdot [\![\tau]\!]_\gamma.\mathsf{size}} \exists v'.\, (p + i \mapsto v') \\
; \ & \text{shr} := \lambda l, \kappa, t, \ell. \\
& \qquad \exists c, p.\, \&_{\mathbf{frac}}^{\kappa}(\lambda q.\, \ell \overset{q}{\mapsto} [l, c, p]) * \\
& \qquad \square(\forall q.\, [\kappa]_q \Rrightarrow \textstyle\ast_{\mathcal{N}_{\mathsf{shr}}, \mathcal{N}_{\mathsf{lft}}}^{\mathcal{N}_{\mathsf{lft}}} \ast_{i=0}^{l} [\![\tau]\!]_\gamma.\mathsf{shr}(\kappa, t, p + i) * [\kappa]_q) \\
\}&
\end{aligned}
$$

C.1.4 Modeling types.

$$
\begin{aligned}
[\![B\langle r\rangle]\!]_\gamma &:= \text{Type}\,\{\ \text{size} := [\![B]\!].\mathsf{size} \ ;\ \text{own} := [\![B]\!].\mathsf{ownown}([\![r]\!]_\gamma) \ ;\ \text{shr} := [\![B]\!].\mathsf{shr}([\![r]\!]_\gamma)\ \} \\
[\![\{v.\ B\langle v\rangle \mid r\}]\!]_\gamma &:= \text{Type}\,\{\ \text{size} := [\![B]\!].\mathsf{size} \\
&\quad ;\ \text{own} := \lambda t, \overline{v}.\, \exists v.\, [\![B]\!].\mathsf{own}([\![r]\!]_{\gamma[v\leftarrow v]}, t, \overline{v}) * [\![r]\!]_{\gamma[v\leftarrow v]} \\
&\quad ;\ \text{shr} := \lambda \kappa, t, \ell.\, \exists v.\, [\![B]\!].\mathsf{shr}([\![r]\!]_{\gamma[v\leftarrow v]}, t, \ell) * [\![r]\!]_{\gamma[v\leftarrow v]}\ \} \\
[\![\&_{\mathbf{shr}}\,\tau]\!]_\gamma &:= \text{SimpleType}(\lambda t, v.\, \exists \ell.\, v = \ell * [\![\tau]\!]_\gamma.\mathsf{shr}([\![\kappa]\!]_\gamma, t, \ell)) \\
[\![\&_{\mathbf{mut}}\,\tau]\!]_\gamma &:= \text{Type}\,\{\ \text{size} := := 1 \\
&\quad ;\ \text{own} := \lambda t, \overline{v}.\, \exists \ell.\, \overline{v} = [\ell] * \&_{\mathbf{full}}^{[\![\kappa]\!]_\gamma} \ell \mapsto [\![\tau]\!]_\gamma.\mathsf{own}(t) \\
&\quad ;\ \text{shr} := \lambda \kappa, t, \ell.\, \exists \ell'.\, \&_{\mathbf{frac}}^{\kappa'}(\lambda q.\, \ell \overset{q}{\mapsto} \ell') * \\
&\qquad \square(\forall q.\, [[\![\kappa]\!]_\gamma \sqcap \kappa']_q \Rrightarrow \textstyle\ast_{\mathcal{N}_{\mathsf{shr}}, \mathcal{N}_{\mathsf{lft}}}^{\mathcal{N}_{\mathsf{lft}}} [\![\tau]\!]_\gamma.\mathsf{shr}([\![\kappa]\!]_\gamma \sqcap \kappa', t, \ell') * [[\![\kappa]\!]_\gamma \sqcap \kappa']_q) \\
&\quad \}\ \\
[\![\mathbf{ptr}(\ell)]\!]_\gamma &:= \text{SimpleType}(\lambda\_, v.\, v = [\![\ell]\!]_\gamma) \\
[\![\lfloor n \rfloor]\!]_\gamma &:= \text{Type}\,\{\ \text{size} := [\![n]\!] \ ;\ \text{own} := \lambda t, \overline{v}.\, |\overline{v}| = [\![n]\!] \\
&\quad ;\ \text{shr} := \lambda \kappa, t, \ell.\ \textstyle\ast_{i=0}^{[\![n]\!]} \exists v.\, \&_{\mathbf{frac}}^{\kappa}(\lambda q.\, \ell + i \overset{q}{\mapsto} v)\ \}
\end{aligned}
$$

C.2 Global environment and function signatures

$$
[\![s]\!]_\gamma : \mathit{Val} \to \mathit{iProp}
$$

$$
\begin{aligned}
[\![\forall \overline{a : \sigma}.\, \mathbf{fn}(r; \overline{x}.\, \mathbf{T}_i) \to \rho_{\mathrm{o}}.\, \mathbf{T}_o]\!]_\gamma(\ell) := \ & \exists f, k, F, q. \\
& \ell \overset{q}{\mapsto} \mathbf{rec}\ f([k] +\!\!+ \overline{x}) \ := \ F * \\
& \triangleright \forall \overline{v_a}, \kappa_{\mathsf{f}}, v_k, \overline{v_x}. \\
& \quad \square[\![\mathbf{E}; \mathsf{f} \sqsubseteq_{\mathrm{l}} [] \mid \mathbf{T}_i; k : \mathbf{cont}(r; \rho_{\mathrm{o}}.\, \mathbf{T}_o) \\
& \quad\ \vdash F[\overline{v_x}/\overline{x}, v_k/k]]\!]_{\gamma[\overline{a}\leftarrow\overline{v_a}][\overline{x}\leftarrow\overline{v_x}][\mathsf{f}\leftarrow\kappa_{\mathsf{f}}][k\leftarrow v_k]}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{\Sigma}]\!]_\gamma &: \mathit{iProp} \\
[\![\emptyset]\!]_\gamma &:= \mathsf{True} \\
[\![\mathbf{\Sigma}, f : s]\!]_\gamma &:= [\![s]\!]_\gamma([\![f]\!]_\gamma) * [\![\mathbf{\Sigma}]\!]_\gamma
\end{aligned}
$$

C.3 Location and continuation contexts

$$
\begin{aligned}
[\![\mathbf{T}]\!] &: \mathit{TId} \to \mathit{iProp} \\
[\![\emptyset]\!]_\gamma(t) &:= \mathsf{True}
\end{aligned}
$$

$$\llbracket \mathbf{T}, \ell \mapsto \tau \rrbracket_\gamma(t) := \llbracket \tau \rrbracket_\gamma.\mathsf{own}(t, [\![\ell]\!]_\gamma]) * \llbracket \mathbf{T} \rrbracket_\gamma(t)$$
$$\llbracket \mathbf{T}, \ell \overset{\dagger\kappa}{\longmapsto} \tau \rrbracket_\gamma(t) := ([\dagger\llbracket\kappa\rrbracket] \Rrightarrow\!\!\!\!\ast_\top \llbracket \tau \rrbracket_\gamma.\mathsf{own}(t, [\![\ell]\!]_\gamma])) * \llbracket \mathbf{T} \rrbracket_\gamma(t)$$

$$\llbracket \mathbf{K} \rrbracket : \mathit{TId} \to \mathit{iProp}$$
$$\llbracket \emptyset \rrbracket(t) := \mathsf{True}$$
$$\llbracket \mathbf{K}, k : \forall \overline{v : \sigma}.\ \mathbf{cont}(\mathbf{L}; \mathbf{T}.\,\rrbracket)(t) := (\forall \overline{v_1}, \overline{v_2}.\ [\mathsf{Na} : t] * \llbracket \mathbf{L} \rrbracket_\gamma(1) * \llbracket \mathbf{T} \rrbracket_{\gamma[\overline{v} \leftarrow \overline{v_1}][\overline{x} \leftarrow \overline{v_2}]}(t) \mathrel{-\!\!*} \mathsf{wp}\ \llbracket k \rrbracket_\gamma(\overline{v_2})\ \{\mathsf{True}\}) \wedge \llbracket \mathbf{K} \rrbracket(t)$$

C.4 Lifetime contexts and judgments

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

$$\llbracket \mathbf{L} \rrbracket_\gamma : \mathit{Frac} \to \mathit{iProp}$$
$$\llbracket \emptyset \rrbracket_\gamma(q) := \mathsf{True}$$
$$\llbracket \mathbf{L}, \kappa \sqsubseteq_\mathrm{l} \overline{\kappa'} \rrbracket_\gamma(q) := \exists \kappa'. \llbracket \kappa \rrbracket_\gamma = \kappa' \sqcap (\prod_{\kappa_i \in \overline{\kappa}} \llbracket \kappa_i \rrbracket_\gamma) * [\kappa']_q * \Box([\kappa'] \Mapsto\!\!\!\!\ast^{\mathcal{N}_\mathrm{lft}}_{\emptyset} [\dagger\kappa']) * \llbracket \mathbf{L} \rrbracket_\gamma(q)$$

$$\llbracket \mathbf{E}_1; \mathbf{L} \vdash \mathbf{E}_2 \rrbracket_\gamma := \forall q.\ \llbracket \mathbf{L} \rrbracket_\gamma(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E}_1 \rrbracket_\gamma \mathrel{-\!\!*} \llbracket \mathbf{E}_2 \rrbracket_\gamma)$$
$$\llbracket \mathbf{E}; \mathbf{L} \vdash \kappa_1 \sqsubseteq \kappa_2 \rrbracket_\gamma := \forall q.\ \llbracket \mathbf{L} \rrbracket_\gamma(q) \mathrel{-\!\!*} \Box(\llbracket \mathbf{E} \rrbracket_\gamma \mathrel{-\!\!*} \llbracket \kappa_1 \rrbracket_\gamma \sqsubseteq \llbracket \kappa_2 \rrbracket_\gamma)$$
$$\llbracket \mathbf{E}; \mathbf{L} \vdash \kappa \text{ alive} \rrbracket_\gamma := \forall q.\ \llbracket \mathbf{E} \rrbracket_\gamma \mathrel{-\!\!*} \langle \llbracket \mathbf{L} \rrbracket_\gamma(q) \Leftrightarrow q'.\, [\llbracket \kappa \rrbracket_\gamma]_{q'} \rangle$$

C.5 Typing Judgments

$$\llbracket \mathbf{\Sigma} \vdash P \rrbracket_\gamma := \bigstar_{D \in P} \llbracket \mathbf{\Sigma} \vdash D \rrbracket_\gamma$$
$$\begin{aligned}\llbracket \mathbf{\Sigma} \vdash \mathbf{fn}\ f(\overline{x})\ \mathbf{ret}\ k\ \{F\} \rrbracket_\gamma := \ & \mathbf{let}\ \forall \overline{v : \sigma}.\ \mathbf{fn}(r; \overline{x}.\ \mathbf{T}_i) \to \rho_o.\ \mathbf{T}_o) = \mathbf{\Sigma}(f)\ \mathbf{in} \\ & \mathbf{let}\ \mathbf{K} = k : \mathbf{cont}(r; \rho_\mathrm{o}.\ \mathbf{T}_o)\ \mathbf{in} \\ & \forall \overline{v_a}, \kappa_\mathsf{f}, v_k, \overline{v_x}. \\ & \quad \llbracket \mathbf{E}; \mathsf{f} \sqsubseteq_\mathrm{l} [\,] \mid \mathbf{T}_i; \mathbf{K} \vdash F[\overline{v_x}/\overline{x}, v_k/k] \rrbracket_{\gamma[\overline{\alpha} \leftarrow \overline{v_a}][\overline{x} \leftarrow \overline{v_x}][\mathsf{f} \leftarrow \kappa_\mathsf{f}][k \leftarrow v_k]}\end{aligned}$$
$$\llbracket \mathbf{E}; \mathbf{L} \mid \mathbf{T}; \mathbf{K} \vdash F \rrbracket_\gamma := \forall t\ \llbracket \mathbf{\Sigma} \rrbracket_\gamma * \llbracket \mathbf{E} \rrbracket_\gamma * \llbracket \mathbf{L} \rrbracket_\gamma * \llbracket \mathbf{T} \rrbracket_\gamma(t) * \llbracket \mathbf{K} \rrbracket_\gamma(t) \mathrel{-\!\!*} \mathsf{wp}\ F\ \{\mathsf{True}\}$$

D SOUNDNESS

Theorem D.1 (Denotational Soundness). *If $\mathbf{\Sigma} \vdash P$, then $\llbracket \mathbf{\Sigma} \vdash P \rrbracket$.*

Theorem D.2 (Stuck Freedom). *If $\llbracket \mathbf{\Sigma} \vdash P \rrbracket$, then the execution of $main$ with the default continuation, i.e., $({}^{*\mathsf{na}} main)(x.x)$, will not end in a stuck state.*