FLUX: Liquid Types for Rust

NICO LEHMANN, UC San Diego, USA
ADAM T. GELLER, University of British Columbia, Canada
NIKI VAZOU, IMDEA, Spain
RANJIT JHALA, UC San Diego, USA

We introduce FLUX, which shows how logical refinements can work hand-in-glove with RUST's ownership mechanisms to yield ergonomic type-based verification of low-level pointer manipulating programs. First, we design a novel refined type system for RUST that indexes mutable locations, with pure (immutable) values that can appear in refinements, and then exploits RUST's ownership mechanisms to abstract sub-structural reasoning about locations within RUST's polymorphic type constructors, while supporting strong updates. We formalize the crucial dependency upon RUST's strong aliasing guarantees by exploiting the *stacked borrows* aliasing model to prove that "well-borrowed evaluations of well-typed programs do not get stuck". Second, we implement our type system in FLUX, a plug-in to the RUST compiler that exploits the factoring of complex invariants into types and refinements to efficiently synthesize loop annotations—including complex quantified invariants describing the contents of containers—via liquid inference. Third, we evaluate FLUX with a benchmark suite of vector manipulating programs and a previously verified secure sandboxing library to demonstrate the advantages of refinement types over program logics as implemented in the state-of-the-art PRUSTI verifier. While PRUSTI's more expressive program logic can, in general, verify *deep* functional correctness specifications, for the *lightweight* but ubiquitous and important verification use-cases covered by our benchmarks, liquid typing makes verification ergonomic by whittling *specification* lines by a factor of two, *verification* time by an order of magnitude, and *annotation* overhead from up to 24% of code size (average 14%), to nothing at all.

CCS Concepts: • **Theory of computation** → **Type structures**; Separation logic; • **Software and its engineering** → *Software verification*.

Additional Key Words and Phrases: Rust, liquid types, heap-manipulating programs

ACM Reference Format:
Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. FLUX: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI, Article 169 (June 2023), 25 pages. https://doi.org/10.1145/3591283

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

Authors' addresses: Nico Lehmann, UC San Diego, USA, nlehmann@ucsd.edu; Adam T. Geller, University of British Columbia, Canada, atgeller@cs.ubc.ca; Niki Vazou, IMDEA, Madrid, Spain, niki.vazou@imdea.org; Ranjit Jhala, UC San Diego, USA, rjhala@ucsd.edu.

2475-1421/2023/6-ART169
https://doi.org/10.1145/3591283

like Prusti [Astrauskas et al. 2019], RustHorn [Matsushita et al. 2021], and Creusot [Denis et al. 2022] 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.

Refinements types 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 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. Previous systems [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 and Formalization (§3) Our first contribution is the design of a type system 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 refinements. Next, Flux shows how to exploit Rust's ownership mechanisms to encapsulate locations, thereby abstracting sub-structural reasoning within Rust's *type constructors*. Finally, Flux extends and refines Rust's mutable references with a notion of *strong references* that precisely track strong updates that alter the type of the mutated object. Crucially, our design relies on the strong aliasing guarantees ensured by Rust without the need to reimplement the complex rules of the borrow checker [Jung et al. 2017; Weiss et al. 2019]. We formalize this requirement by defining an operational semantics instrumented with a "dynamic borrow checker" as defined by the *Stacked Borrows* aliasing discipline [Jung et al. 2019]. Armed with this dynamic interpretation of Rust's aliasing model we prove soundness of our type system, which ensures that *"well-borrowed evaluations of well-typed programs do not get stuck"* (Theorem 3.1).

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

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

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

Fig. 1. Examples showing Flux basic features: indexed types, existential types and refinement parameters.

program is well-typed. Finally, in the third *inference* phase, we solve the constraints, using the fixpoint implementation of Cosman and Jhala [2017], and either 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.

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, and security properties in parts of a previously verified sandboxing library. Prusti's program logic can, in general, verify *deep* functional correctness specifications beyond the scope of Flux. However, for the ubiquitous and important *lightweight* verification use cases exemplified by our benchmarks, our evaluation shows how Flux's refined types naturally capture invariants and heap update specifications that must otherwise be spelled out via complex (quantified) program logic assertions. Consequently, we show how liquid typing makes lightweight verification ergonomic by whittling verification time by an order of magnitude, specification sizes by a factor of 2, and the loop-invariant annotation overhead from up to 24% of code size (average 14%), to nothing at all.

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 *ownership types* allows us to precisely track refinements in the presence of imperative mutation (§2.2). Finally, we show how the combination of ownership and refinement types enables ergonomic verification, by looking at some examples that work over unbounded collections (§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 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

[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`.

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]`. Indices do not always encode singletons. As an example, the type `RVec<T>[n]` of *growable* vectors is indexed by their length (of sort `int`), as detailed later in §2.3. Even though most of our examples have a single index, types can have multiple indices. For example, in §5 we index a type for 2-D matrices by both 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 run-time. 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 refined signature for the function `is_pos` specified with the attribute `#[flux::sig(...)]` on the left in fig. 1. The function `is_pos` tests whether a 32-bit signed integer is 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 `n > 0`. The syntax `@n` is used to bind and quantify over `n` for the scope of the function.

Existential Types Indexed types suffice when we know the exact value of the underlying term, *i.e.,* we 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 accommodates such specifications via *existential types* of the form `{v. B[v] | p}` where: `v` is a variable in the refinement logic, `B[v]` is a base type indexed by `v`, and `p` is a predicate constraining `v`. For example, the existential type `{v. i32[v] | v > 0}` specifies the set of *positive* 32-bit integers. Similarly, the set of *non-empty* vectors is described by the type `{v. RVec<T>[v] | v > 0}`. We define the syntax `B{v: p}` to mean `{v. B[v] | p}`. Hence, the two types above abbreviate to `i32{v: v > 0}` and `RVec<T>{v: v > 0}`. Further, we write `B` to abbreviate `B{v: true}` and `nat` to abbreviate `i32{v: v >= 0}`.

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`.[2] 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 as `x`.

2.2 Ownership

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.

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 values at that location at different points in time. For example, if a variable `x` has type `i32[n]`, after executing the statement `x += 1`, the type of `x` is updated to `i32[n + 1]`.

Borrowing Exclusive ownership suffices for local updates but for more complex data, functions must eventually relinquish ownership to other functions that update and read the data in some fashion. Rust's unique approach to allow this is called *borrowing*, via two kinds of references that

[2]The attentive reader will note that the implementation of `abs` causes an overflow if `x` equals `i32::MIN`. Flux can easily verify the absence of overflows statically, but to keep the examples short we assume that overflows are being checked at run-time which is enabled in the compiler with the flag `-C overflow-checks=yes`.

```
1 #[flux::sig(fn(&mut nat))]
2 fn decr(x: &mut i32) {
3     let y = *x;
4     if y > 0 {
5         *x = y - 1;
6     }
7 }
8
9 #[flux::sig(fn(bool) -> nat)]
10 fn ref_join(z: bool) -> i32 {
11     let mut x = 1;
12     let mut y = 2;
13     let r = if z { &mut x } else { &mut y };
14     decr(r);
15     x
16 }
17 fn swap<T>(x: &mut T, y: &mut T);
18
19 #[flux::sig(fn() -> nat)]
20 fn use_swap() -> i32 {
21     let mut x = 0;
22     let mut y = 1;
23     swap(&mut x, &mut y);
24     x
25 }
26
27 #[flux::sig(
28   fn(x: &strg i32[@n])
29   ensures *x: i32[n + 1])]
30 fn incr(x: &mut i32) {
31     *x += 1;
32 }
```

Fig. 2. Examples showing the interaction between refinement types and ownership types.

grant temporary access to a memory location. 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 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 one mutable reference to a value at a time.

Flux exploits the semantics of mutable references to attach invariants to data. Crucially, updates through a mutable reference `&mut T` *do not change* the type `T`, or in other words, mutating through a mutable reference can only perform *weak updates*. This behavior ensures that mutations through an `&mut T` will preserve the invariants encoded in `T`. For example, consider the function `decr` in fig. 2, whose plain rust signature is `fn(&mut i32) -> ()`. (Hereafter, we follow the standard Rust style and omit the return type if it is the unit type `()`.) The Flux signature takes as input an `&mut nat` (*i.e.,* `&mut i32{v: v >= 0}`) imposing on the function the obligation to preserve the invariant that the reference points to a natural number. This means that the update in line 5 must preserve the type, which Flux can prove assuming the condition in the branch.

Imprecise Alias Information A mutable reference will typically point to a memory location that cannot be determined statically. Still, we would like to track refinements on the locations that *may* be pointed to by a reference. Next, we show how Flux leverages Rust's borrowing rules to track refinements in the presence of imprecise aliasing information.

Consider the function `ref_join` in fig. 2. The syntax `&mut x` (*resp.* `&mut y`) in line 13 is used to create a mutable reference by temporarily borrowing the content of `x` (*resp.* `y`). Then, depending on the branch condition, `r` will point to either `x` or `y`. Acknowledging the reference may end up pointing to an unknown location, when borrowing `x`, Flux updates its type to account for possible mutations through `r`, which in turn, must only allow updates guaranteeing `x` will continue to have this type after the borrow ends. Concretely, Flux updates the type of `x` to be `nat`, and assigns `r` the type `&mut nat`. When the borrow expires after line 14, we can read again from `x` knowing it is still a natural number. Note that at the time `x` is borrowed in line 13, Flux does not know immediately what type should be assigned to `x` as the appropriate type depends on subsequent uses of `x` and `r`. In §3 and §4 we *resp.* show how Flux's liquid typing can check and automatically infer this type.

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

Fig. 3. A refined API for vectors indexed by their size.

Specs for Free via Polymorphism For a classical type system, polymorphism facilitates code *reuse*: we can use the same datatype to hold integers or strings or booleans *etc.*. Flux exploits the combination of polymorphism and mutable references to generate compact specifications. Consider the function `use_swap` in fig. 2 which uses the function `swap` from the Rust standard library to swap the values of `x` and `y`. The plain Rust signature of `swap` is `fn<T>(&mut T, &mut T)` where `T` is a polymorphic type parameter. Just using the plain Rust signature—and no other specifications—Flux can verify the post-condition of `use_swap` by automatically instantiating the parameter `T` to be `nat` via liquid typing. After the function returns, `x` and `y` are guaranteed to have type `nat` because by virtue of taking `&mut T` references, `swap` will respect the invariants in `T`.

Strong updates We have seen how the mechanisms Rust uses to control mutation interact with refinements. Exclusive ownership provides local strong updates, *i.e.,* within the function owning a value, and mutable references can be used to temporarily relinquish ownership and provide weak updates while preserving the ability to track refinements in the presence of imprecise alias information. While powerful, these mechanisms are insufficient for refinement type checking.

In many situations, we would like to lend a value to other functions that *change* the value's refinement upon their return. 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 by tracking the precise location the reference points to. Flux accommodates strong references by extending function signatures to specify the *updated type* of each strong reference after the function returns. For example, consider the signature of `incr` in fig. 2. The Flux signature refines the plain Rust signature to specify that (1) the argument `x` is a *strong* reference to an `i32[n]` and (2) the updated type of the location pointed to by `x` is `i32[n + 1]` as denoted by the function's `ensures` clause. With this specification, Flux can verify that after executing the statements `let mut x = 1; incr(&mut x)` the type of `x` is `i32[2]`.

2.3 Unbounded Collections

Next, we illustrate how the fundamental mechanisms introduced so far enable ergonomic verification by showing how they can be used to automatically verify lightweight properties about unbounded collections. First, we present a refined API for *vectors*. Second, we use this API to concisely specify fragments of an implementation of the k-means clustering algorithm. Finally, we present a refined implementation of a *linked list*, to illustrate how standard type constructors can be used 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.

A Refined Vector API Fig. 3 summarizes the signatures for `RVec`—vectors refined by their size.

- `new` constructs empty vectors: the return type `RVec<T>[0]` states the returned vector has size `0`.

- `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 type `usize[n]` stipulates that the result equals the receiver's size.
- `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. 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, which can point to a possible unbounded set of locations, 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` is used to grow the vector by one element at a time. It 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]`.

Constructing a Vector Fig. 4 shows a couple of functions taken from an implementation of the k-means clustering algorithm. The function `init_zeros` takes as input a `usize` equal to `n` and returns as output an `n`-dimensional vector of 32-bit floats (`f32`), specified as `RVec<f32>[n]`. The variable `vec` is initialized with an empty vector using the function `new` in line 5. Similarly, the counter `i` is initialized with `0` in line 6. In each iteration, the method `push` is called on `vec` incrementing its size by one. Correspondingly, the counter `i` is also incremented by one. 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]`.

Quantified Invariants via Polymorphism `RVec` is polymorphic over `T`: the type of elements it contains. We can instantiate `T` with arbitrary refined types, which is exploited by FLUX to compactly specify that *all* elements of the vector satisfy some invariant. For instance, the function `normalize_centers` in fig. 4 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 mutable references to these vectors to the function `normal` in line 25.

A Refined Linked List Fig. 5 shows a standard definition of a recursive `List` using an `enum` which is RUST's syntax to declare algebraic data types. As required by RUST, each recursive occurrence of the type needs to be guarded by a pointer to ensure the size is known at compile time. We use the standard type `Box<T>`, which represents an owned (heap-allocated) pointer to values of type `T`. The annotation `#[flux::refined_by(len: int)]` on top of the `enum` declares that the type is indexed by an integer in the logic, which we mean to represent the length of the list. Each variant is annotated with the attribute `#[flux::variant(...)]` to specify a refined signature for the constructor. We define the `Nil` case to return a `List<T>[0]`, declaring its length to be zero. In the `Cons` case, given a value of type `T` and a `List` of length `n` (inside a `Box`) the constructor returns a list of length `n + 1` as declared by the return type `List<T>[n + 1]`.

Finally, we show how the indexed length can be used to specify the method `append` at the bottom of fig. 5. The method takes two lists of length `n` and `m`, consuming the second one and appending it to the end of the first one *in place*, *i.e.,* using RUST's idioms to avoid copying. The `ensures *self: List<T>[n + m]` clause specifies that the length of the first list gets updated

```
#[flux::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
}
```

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

Fig. 4. Code taken from an implementation of the k-means clustering algorithm. The code uses RVec to represent k-centers of n-dimensional points.

```
#[flux::refined_by(len: int)]
enum List<T> {
   #[flux::variant(List<T>[0])]
   Nil,
   #[flux::variant((T, Box<List<T>[@n]>) -> List<T>[n + 1])]
   Cons(T, Box<List<T>>)
}

impl<T> List<T> {
   #[flux::sig(fn(self: &strg List<T>[@n], List<T>[@m]) ensures *self: List<T>[n+m])]
   fn append(&mut self, other: List<T>) {
       match self {
           List::Cons(_, tl) => tl.append(other),
           List::Nil => *self = other,
       }
   }
}
```

Fig. 5. Implementation of a refined linked list.

to `n + m`. The implementation recursively matches on the list `self` until `Nil` is found, at which point `self` is updated in place to point to `other`. Using standard syntax directed typing rules, Flux decomposes the verification into a (quantifier free) *verification condition* of the form:

$$(0 = n \Rightarrow m = n + m) \wedge (v + 1 = n \Rightarrow v + m + 1 = n + m)$$

where the first conjunct checks that `*self` has type `List<T>[m]` after the update in the *base case*, and the second checks that `*self` has type `List<T>[n + m]` after the *recursive call*.

$$
\begin{array}{rlcl}
\textbf{\textit{Refinements}} & r & ::= & a \mid \ell \mid \textbf{true} \mid \textbf{false} \mid 0, \pm 1, \ldots \mid r = r \mid \neg r \mid r\ [\wedge, \vee]\ r \mid r\ [+, -, *]\ r \\
\textbf{\textit{Expressions}} & e & ::= & \textbf{let}\ x = \textbf{new}(\rho)\ \textbf{in}\ e \mid \textbf{unpack}(x, a)\ \textbf{in}\ e \mid \textbf{call}\ e[\overline{r}](\overline{av}) \mid p := e \\
 & & \mid & \textbf{if}\ e\ \{e\}\ \textbf{else}\ \{e\} \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \&\textbf{strg}\ p \mid \&\textbf{mut}\ p \mid \&\textbf{shr}\ p \mid *p \mid x \mid v \\
\textbf{\textit{Values}} & v & ::= & \textbf{rec}\ f[\overline{a}](\overline{x}) \ := \ e \mid \textbf{true} \mid \textbf{false} \mid 0, \pm 1, \ldots \mid \text{☣} \mid \mathrm{ptr}(\ell, t) \\
\textbf{\textit{A-values}} & av & ::= & x \mid v \\
\textbf{\textit{Places}} & p & ::= & x \mid \mathrm{ptr}(\ell, t) \\
\textbf{\textit{Types}} & \tau & := & B[r] \mid \{a.\ B[a] \mid r\} \mid \textbf{ptr}(\eta) \mid \&_{\mu}\ \tau \mid \lightning \mid \forall \overline{a : \sigma}.\ \textbf{fn}(\mathrm{T}; \overline{\tau}) \to \tau / \mathrm{T} \\
\textbf{\textit{Base Types}} & B & ::= & \textbf{int} \mid \textbf{bool} \\
\textbf{\textit{Modifier}} & \mu & ::= & \textbf{mut} \mid \textbf{shr} \\
\textbf{\textit{Locations}} & \eta & := & \ell \mid \rho \\
\textbf{\textit{Sorts}} & \sigma & := & \textbf{int} \mid \textbf{bool} \mid \textbf{loc}
\end{array}
\qquad
\begin{array}{rlcl}
\textbf{\textit{Contexts}} & & & \\
\textbf{\textit{Value}} & \Gamma & := & \emptyset \mid \Gamma, x : \tau \\
\textbf{\textit{Refinement}} & \Delta & := & \emptyset \mid \Delta, a : \sigma \mid \Delta, r \\
\textbf{\textit{Location}} & \mathrm{T} & := & \emptyset \mid \mathrm{T}, \eta \mapsto \tau
\end{array}
$$

Fig. 6. Syntax of λ_{LR}.

3 FORMALIZATION

In this section, we introduce λ_{LR}, a core calculus which models RUST's safe fragment extended with refinement types. To aid understanding, we first describe the syntax (§3.1) and type system (§3.2) using only simple data types (**int** and **bool**). Next, we show how to extend the system with vectors (§3.3). Crucially, we define λ_{LR}'s type system as an analysis to be layered on top of RUST's ownership system. Instead of relying on the details of the borrow checker, we capture this requirement by instrumenting the operational semantics with a dynamic analysis based on the *Stacked Borrows* aliasing discipline [Jung et al. 2019] and use it to prove soundness of λ_{LR}'s type system (§3.4). The complete definitions and proofs can be found in our technical appendix [Lehmann et al. 2023b].

3.1 Syntax of λ_{LR}.

Fig. 6 summarizes the syntax of λ_{LR}. Most of the grammar is based on a standard call-by-value language with (RUST-like) references. In the following we discuss the bits that are different.

Refinements The language of logical refinements includes refinement variables, constants for booleans and integers, and operations for equality, boolean logic, and integer arithmetic. We write refinement variables as a in general and (by convention) as ρ when referring to an abstract location. Additionally, refinements contain concrete locations ℓ which show up due to the operational rules.

Expressions Local variables, introduced with **let**-bindings and written x or f, are pure values. This differs from RUST's local variables which are mutable and addressable. To model RUST's variables correctly, we use $\textbf{let}\ x = \textbf{new}(\rho)\ \textbf{in}\ e$ to bind the local variable x to a *heap-allocated* location represented by the variable ρ.

A function is declared as $\textbf{rec}\ f[\overline{a}](\overline{x}) \ := \ e$, where f is a binder for the (potentially) recursive call, $\overline{a}$ is a list of refinement parameters, and $\overline{x}$ a list of binders for the arguments. Functions can be called using $\textbf{call}\ e[\overline{r}](\overline{av})$, where $\overline{r}$ is the list used to instantiate refinement parameters and $\overline{av}$ is a list of arguments. Arguments must be *A-values* (either x or v), which simplifies the typing rules.

The $\textbf{unpack}(x, a)\ \textbf{in}\ e$ instruction is used when a variable x has type $\{b.\ B[b] \mid r\}$ to introduce a fresh name a for the (existentially quantified) refinement variable b.

In addition to RUST's $\&\textbf{shr}\ p$ and $\&\textbf{mut}\ p$ borrow expressions, λ_{LR} includes $\&\textbf{strg}\ p$ to borrow a strong pointer. Borrows are restricted over a place p which can be either a variable or a *tagged pointer* $\mathrm{ptr}(\ell, t)$. Tagged pointers only show up at run-time and are discussed in §3.4. The same place-only restriction applies to the left-hand side of an assignment $p := e$ and to dereferences $*p$.

A *poison value*, ☣, is used to represent uninitialized memory and will cause the program to get stuck if used in any way that affects the evaluation (*e.g.*, as a branch condition). We also use ☣ to evaluate an expression when the value is not relevant, *e.g.*, the value returned by an assignment.

$$
\begin{array}{l}
\texttt{decr} : \mathrm{fn}(\&_{\mathrm{mut}}\ \mathrm{nat}) \rightarrow \lightning \\
\textbf{let}\ \texttt{decr} = \textbf{rec}\ f[](x)\ := \\
\quad \Delta_1 = \emptyset; \\
\quad \Gamma_1 = \texttt{gt}: ..., \texttt{sub}: ..., \\
\qquad\quad f: ..., x: \&_{\mathrm{mut}}\ \mathrm{nat}; \\
\quad \mathrm{T}_1 = \emptyset \\
\quad \textbf{let}\ y = *x\ \textbf{in} \\
\quad \Delta_2 = \emptyset; \Gamma_2 = \Gamma_1, y: \mathrm{nat}; \mathrm{T}_2 = \mathrm{T}_1 \\
\quad \textbf{unpack}\ (y, a_y)\ \textbf{in} \\
\quad \Delta_3 = a_y: \mathrm{int}, a_y \geq 0; \\
\quad \Gamma_3 = \Gamma_1, y: \mathrm{int}[a_y]; \\
\quad \mathrm{T}_3 = \mathrm{T}_1 \\
\quad \textbf{if call}\ \texttt{gt}[a_y, 0](y, 0)\ \{ \\
\qquad \Delta_4 = \Delta_3, a_y > 0; \\
\qquad \Gamma_4 = \Gamma_3; \\
\qquad \mathrm{T}_4 = \mathrm{T}_1 \\
\qquad x := \textbf{call}\ \texttt{sub}[a_y, 1](y, 1) \\
\qquad \textsc{T-Ass} : \Delta_4 \vdash \mathrm{int}[a_y - 1] \preccurlyeq \mathrm{nat} \\
\quad \}\ \textbf{else}\ \{☣\}
\end{array}
$$

$$
\begin{array}{l}
\texttt{ref\_join} : \forall a{:}\mathrm{bool}.\ \mathrm{fn}(\mathrm{bool}[a]) \rightarrow \mathrm{nat} \\
\textbf{let}\ \texttt{ref\_join} = \textbf{rec}\ f[a](z)\ := \\
\quad \Delta_1 = a{:}\mathrm{bool}; \Gamma_1 = \texttt{decr}{:}\ldots, f{:}\ldots, z{:}\mathrm{bool}[a]; \mathrm{T}_1 = \emptyset \\
\quad \textbf{let}\ x = \textbf{new}(\rho_x)\ \textbf{in} \\
\quad \Delta_2 = \Delta_1, \rho_x{:}\mathrm{loc}; \Gamma_2 = \Gamma_1, x{:}\mathrm{ptr}(\rho_x); \mathrm{T}_2 = \mathrm{T}_1, \rho_x \mapsto \lightning \\
\quad x := 1; \\
\quad \Delta_3 = \Delta_2; \Gamma_3 = \Gamma_2; \mathrm{T}_3 = \mathrm{T}_1, \rho_x \mapsto \mathrm{int}[1] \\
\quad \textbf{let}\ y = \textbf{new}(\rho_y)\ \textbf{in}\ y := 2; \\
\quad \Delta_4 = \Delta_3, \rho_y{:}\mathrm{loc}; \Gamma_4 = \Gamma_3, y{:}\mathrm{ptr}(\rho_y); \mathrm{T}_4 = \mathrm{T}_3, \rho_y \mapsto \mathrm{int}[2] \\
\quad \textbf{let}\ r = \textbf{if}\ z\ \{ \\
\qquad \Delta_{5_1} = \Delta_4, a; \Gamma_{5_1} = \Gamma_4; \mathrm{T}_{5_1} = \mathrm{T}_4 \\
\qquad \&\textbf{mut}\ x \quad : \&_{\mathrm{mut}}\mathrm{nat}, \text{by rule } \textsc{T-bsmut} \\
\qquad \Delta_{6_1} = \Delta_{5_1}; \Gamma_{6_1} = \Gamma_4; \mathrm{T}_{6_1} = \rho_x \mapsto \mathrm{nat}, \rho_y \mapsto \mathrm{int}[2] \\
\quad \}\ \textbf{else}\ \{ \\
\qquad \Delta_{5_2} = \Delta_4, \neg a; \Gamma_{5_2} = \Gamma_4; \mathrm{T}_{5_2} = \mathrm{T}_4 \\
\qquad \&\textbf{mut}\ y \quad : \&_{\mathrm{mut}}\mathrm{nat}, \text{by rule } \textsc{T-bsmut} \\
\qquad \Delta_{6_2} = \Delta_{5_2}; \Gamma_{6_2} = \Gamma_4; \mathrm{T}_{6_2} = \rho_x \mapsto \mathrm{int}[1], \rho_y \mapsto \mathrm{nat} \\
\quad \}\ \ \Delta_7 = \Delta_4; \Gamma_7 = \Gamma_4; \mathrm{T}_7 = \rho_x \mapsto \mathrm{nat}, \rho_y \mapsto \mathrm{nat} \\
\quad \textbf{in} \\
\quad \Delta_8 = \Delta_7; \Gamma_8 = \Gamma_7, r{:}\&_{\mathrm{mut}}\mathrm{nat}; \mathrm{T}_8 = \mathrm{T}_7 \\
\quad \textbf{call}\ \texttt{decr}(r); *x \quad : \mathrm{nat}, \text{by } \textsc{T-deref-strg}
\end{array}
$$

Fig. 7. λ_{LR} encoding and type checking of the examples `decr` (left) and `ref_join` (right) from § 2.

Types As discussed in §2.1, *indexed types* $B[r]$ and *existential types* $\{a.\ B[a] \mid r\}$ refine a base type B, which can be either **int** or **bool**. Next, the type $\lightning$ describes uninitialized memory (☣).

There are two kinds of pointer types: *strong pointers* $\mathbf{ptr}(\eta)$ and *(borrowed) references* $\&_\mu\tau$. A strong pointer $\mathbf{ptr}(\eta)$ points to a precise location η (either concrete ℓ or abstract ρ). Because we model the stack using heap allocations, strong pointers also represent Rust's local variables unifying the treatment of exclusive ownership and strong references discussed in §2.2. References $\&_\mu\tau$, which represent standard Rust references, are qualified by a *modifier* μ which can be either **shr** (for shared references) or **mut** (for mutable references).

A function type $\forall\overline{a{:}\sigma}.\ \mathbf{fn}(\mathrm{T}_i;\overline{\tau}) \rightarrow \tau/\mathrm{T}_o$, can be parameterized by a list of refinement variables $\overline{a}$ each with a declared sort σ. The location contexts T_i and T_o capture the type of locations before and after the function call. For example,

$$\forall a{:}\mathbf{int}, \rho{:}\mathbf{loc}.\ \mathbf{fn}(\rho \mapsto \mathbf{int}[a]; \mathbf{ptr}(\rho)) \rightarrow \lightning/\rho \mapsto \mathbf{int}[a+1]$$

is the type of a function that takes a strong pointer to an $\mathbf{int}[a]$ and updates it to $\mathbf{int}[a+1]$ (the type of `incr` in fig. 2). We omit the list of refinement parameters, the list of arguments, or the input and output location contexts if they are empty.

3.2 Type Checking of λ_{LR}.

Figs. 8 and 9 define the three main judgments of λ_{LR}. They use three kinds of contexts (fig. 6). The *refinement context* Δ maps refinement variables to sorts and also contains predicates that relate these variables. The *value context* Γ tracks local variables in scope and maps them to types. Finally, the *location context* T describes ownership of locations with their corresponding types.

The typing judgment $\Delta; \Gamma; \mathrm{T}_i \vdash e : \tau \dashv \mathrm{T}_o$ states that under the refinement context Δ, value context Γ, and input location context T_i, the expression e has type τ and produces a location context T_o. The output type and location context can respectively be weakened using the judgments for subtyping ($\Delta \vdash \tau_1 \preccurlyeq \tau_2$) and location context inclusion ($\Delta \vdash \mathrm{T}_1 \Rightarrow \mathrm{T}_2$).

Expression Typing $\boxed{\Delta;\Gamma;\mathrm{T}\vdash e:\tau\dashv\mathrm{T}}$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathrm{T}_i\vdash e:\tau_1\dashv\mathrm{T}\\ \Delta\vdash\tau_1\preccurlyeq\tau\qquad\Delta\vdash\mathrm{T}\Rightarrow\mathrm{T}_o\end{array}}{\Delta;\Gamma;\mathrm{T}_i\vdash e:\tau\dashv\mathrm{T}_o}\,\textsc{T-sub}\qquad\frac{\begin{array}{c}\Delta\vdash_{\mathtt{wf}}\tau\qquad\Delta\vdash_{\mathtt{wf}}\mathrm{T}_o\\ \Delta,\rho:\mathbf{loc};\Gamma,x:\mathbf{ptr}(\rho);\mathrm{T}_i,\rho\mapsto\lightning\vdash e:\tau\dashv\mathrm{T}_o\end{array}}{\Delta;\Gamma;\mathrm{T}_i\vdash\mathbf{let}\ x=\mathbf{new}(\rho)\ \mathbf{in}\ e:\tau\dashv\mathrm{T}_o}\,\textsc{T-new}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathrm{T}_i\vdash e_x:\tau_x\dashv\mathrm{T}\\ \Delta;\Gamma,x:\tau_x;\mathrm{T}\vdash e:\tau\dashv\mathrm{T}_o\end{array}}{\Delta;\Gamma;\mathrm{T}_i\vdash\mathbf{let}\ x=e_x\ \mathbf{in}\ e:\tau\dashv\mathrm{T}_o}\,\textsc{T-let}\qquad\frac{\begin{array}{c}\Delta;\Gamma;\mathrm{T}_i\vdash e:\mathbf{bool}[r]\dashv\mathrm{T}_o\\ \Delta,r;\Gamma;\mathrm{T}_o\vdash e_1:\tau\dashv\mathrm{T}\\ \Delta,\neg r;\Gamma;\mathrm{T}_o\vdash e_2:\tau\dashv\mathrm{T}\end{array}}{\Delta;\Gamma;\mathrm{T}_i\vdash\mathbf{if}\ e\ \{e_1\}\ \mathbf{else}\ \{e_2\}:\tau\dashv\mathrm{T}}\,\textsc{T-if}$$

$$\frac{\begin{array}{c}\forall i.\Delta;\Gamma;\mathrm{T}\vdash av_i:\theta\cdot\tau_i\dashv\mathrm{T}\qquad\Delta;\Gamma;\mathrm{T}\vdash e:\forall\overline{a:\sigma}.\ \mathbf{fn}(\mathrm{T}_i;\overline{\tau})\rightarrow\tau_o/\mathrm{T}_o\dashv\mathrm{T}_1,\mathrm{T}_2\\ \theta=[\overline{r}/\overline{a}]\qquad\Delta\vdash\mathrm{T}_1\Rightarrow\theta\cdot\mathrm{T}_i\qquad\forall i.\Delta\vdash r_i:\sigma_i\end{array}}{\Delta;\Gamma;\mathrm{T}\vdash\mathbf{call}\ e[\overline{r}](\overline{av}):\theta\cdot\tau_o\dashv\theta\cdot\mathrm{T}_o,\mathrm{T}_2}\,\textsc{T-call}$$

$$\frac{\Delta,a:\mathtt{sort}(B),r;\Gamma_1,x:B[a],\Gamma_2;\mathrm{T}_i\vdash e:\tau\dashv\mathrm{T}_o}{\Delta;\Gamma_1,x:\{a.\ B[a]\mid r\},\Gamma_2;\mathrm{T}_i\vdash\mathbf{unpack}(x,a)\ \mathbf{in}\ e:\tau\dashv\mathrm{T}_o}\,\textsc{T-unpack}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\mathrm{T}_o\vdash p:\&_{\mathbf{mut}}\tau\dashv\mathrm{T}_o\\ \Delta;\Gamma;\mathrm{T}_i\vdash e:\tau_v\dashv\mathrm{T}_o\qquad\Delta\vdash\tau_v\preccurlyeq\tau\end{array}}{\Delta;\Gamma;\mathrm{T}_i\vdash p:=e:\lightning\dashv\mathrm{T}_o}\,\textsc{T-ass}\qquad\frac{\begin{array}{c}\Delta;\Gamma;\mathrm{T}_o\vdash p:\mathbf{ptr}(\eta)\dashv\mathrm{T}_o\\ \Delta;\Gamma;\mathrm{T}_i\vdash e:\tau\dashv\mathrm{T}_o\end{array}}{\Delta;\Gamma;\mathrm{T}_i\vdash p:=e:\lightning\dashv\mathrm{T}_o[\eta\mapsto\tau]}\,\textsc{T-ass-strg}$$

Values

$$\frac{\Delta,\overline{a:\sigma};\Gamma,\overline{x:\tau},f:\forall\overline{a:\sigma}.\ \mathbf{fn}(\mathrm{T}_i;\overline{\tau})\rightarrow\tau/\mathrm{T}_o;\mathrm{T}_i\vdash e:\tau\dashv\mathrm{T}_o}{\Delta;\Gamma;\mathrm{T}\vdash\mathbf{rec}\ f[\overline{a}](\overline{x})\ :=\ e:\forall\overline{a:\sigma}.\ \mathbf{fn}(\mathrm{T}_i;\overline{\tau})\rightarrow\tau/\mathrm{T}_o\dashv\mathrm{T}}\,\textsc{T-fun}\qquad\frac{x:\tau\in\Gamma}{\Delta;\Gamma;\mathrm{T}\vdash x:\tau\dashv\mathrm{T}}\,\textsc{T-var}$$

$$\frac{c\in\{\mathbf{true},\mathbf{false}\}}{\Delta;\Gamma;\mathrm{T}\vdash c:\mathbf{bool}[c]\dashv\mathrm{T}}\,\textsc{T-bool}\qquad\frac{}{\Delta;\Gamma;\mathrm{T}\vdash\Biohazard:\lightning\dashv\mathrm{T}}\,\textsc{T-mem}\qquad\frac{}{\Delta;\Gamma;\mathrm{T}\vdash i:\mathbf{int}[i]\dashv\mathrm{T}}\,\textsc{T-int}$$

Borrows

$$\frac{\Delta;\Gamma;\mathrm{T}\vdash p:\mathbf{ptr}(\eta)\dashv\mathrm{T}}{\Delta;\Gamma;\mathrm{T}\vdash\&\mathbf{strg}\ p:\mathbf{ptr}(\eta)\dashv\mathrm{T}}\,\textsc{T-bstrg}\qquad\frac{\Delta;\Gamma;\mathrm{T}\vdash p:\&_{\mathbf{mut}}\tau\dashv\mathrm{T}}{\Delta;\Gamma;\mathrm{T}\vdash\&\mathbf{mut}\ p:\&_{\mathbf{mut}}\tau\dashv\mathrm{T}}\,\textsc{T-bmut}$$

$$\frac{\begin{array}{c}\Delta\vdash\mathrm{T}(\eta)\preccurlyeq\tau\\ \Delta;\Gamma;\mathrm{T}\vdash p:\mathbf{ptr}(\eta)\dashv\mathrm{T}\end{array}}{\Delta;\Gamma;\mathrm{T}\vdash\&\mathbf{mut}\ p:\&_{\mathbf{mut}}\tau\dashv\mathrm{T}[\eta\mapsto\tau]}\,\textsc{T-bsmut}\qquad\frac{\begin{array}{c}\Delta\vdash\tau'\preccurlyeq\tau\\ \Delta;\Gamma;\mathrm{T}\vdash p:\&_{\mu}\tau'\dashv\mathrm{T}\end{array}}{\Delta;\Gamma;\mathrm{T}\vdash\&\mathbf{shr}\ p:\&_{\mathbf{shr}}\tau\dashv\mathrm{T}}\,\textsc{T-bshr}$$

Dereference

$$\frac{\Delta;\Gamma;\mathrm{T}\vdash p:\&_{\mu}\tau\dashv\mathrm{T}}{\Delta;\Gamma;\mathrm{T}\vdash *p:\tau\dashv\mathrm{T}}\,\textsc{T-deref}\qquad\frac{\Delta;\Gamma;\mathrm{T}\vdash p:\mathbf{ptr}(\eta)\dashv\mathrm{T}}{\Delta;\Gamma;\mathrm{T}\vdash *p:\mathrm{T}(\eta)\dashv\mathrm{T}}\,\textsc{T-deref-strg}$$

Fig. 8. Expression typing of λ_{LR} (some well-formedness requirements are omitted).

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

$$\frac{\Delta \vdash \mathrm{T}_1 \Rightarrow \mathrm{T}_2 \qquad \Delta \vdash \mathrm{T}_2 \Rightarrow \mathrm{T}_3}{\Delta \vdash \mathrm{T}_1 \Rightarrow \mathrm{T}_3}\textsc{C-trans} \qquad \frac{\mathrm{T}' \text{ is a permutation of } \mathrm{T}}{\Delta \vdash \mathrm{T} \Rightarrow \mathrm{T}'}\textsc{C-Perm}$$

$$\frac{}{\Delta \vdash \mathrm{T}, \mathrm{T}' \Rightarrow \mathrm{T}}\textsc{C-Weak} \qquad \frac{\Delta \vdash \mathrm{T}_1 \Rightarrow \mathrm{T}_2}{\Delta \vdash \mathrm{T}, \mathrm{T}_1 \Rightarrow \mathrm{T}, \mathrm{T}_2}\textsc{C-Frame} \qquad \frac{\Delta \vdash \tau_1 \preccurlyeq \tau_2}{\Delta \vdash \eta \mapsto \tau_1 \Rightarrow \eta \mapsto \tau_2}\textsc{C-Sub}$$

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

$$\frac{}{\Delta \vdash \mathbf{ptr}(\eta) \preccurlyeq \mathbf{ptr}(\eta)}\textsc{S-ptr} \qquad \frac{}{\Delta \vdash \lightning \preccurlyeq \lightning}\textsc{S-mem} \qquad \frac{\Delta \models r_1 = r_2}{\Delta \vdash B[r_1] \preccurlyeq B[r_2]}\textsc{S-idx}$$

$$\frac{\Delta, a : \mathtt{sort}(B), r \vdash B[a] \preccurlyeq \tau}{\Delta \vdash \{a.\ B[a] \mid r\} \preccurlyeq \tau}\textsc{S-unpack} \qquad \frac{\Delta \models r_2[r_1/a]}{\Delta \vdash B[r_1] \preccurlyeq \{a.\ B[a] \mid r_2\}}\textsc{S-ex}$$

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

$$\frac{\begin{array}{c}\Delta, \overline{a : \sigma} \models r_2 \Rightarrow r_1 \qquad \Delta, \overline{a : \sigma} \vdash \mathrm{T}_{2i} \Rightarrow \mathrm{T}_{1i} \qquad \forall i. \Delta, \overline{a : \sigma} \vdash \tau_{2i} \preccurlyeq \tau_{1i} \\ \Delta, \overline{a : \sigma} \vdash \mathrm{T}_{1o} \Rightarrow \mathrm{T}_{2o} \qquad \Delta, \overline{a : \sigma} \vdash \tau_{1o} \preccurlyeq \tau_{2o}\end{array}}{\Delta \vdash \forall \overline{a : \sigma}.\ \mathbf{fn}(\mathrm{T}_{1i}; \overline{\tau_1}) \to \tau_{1o}/\mathrm{T}_{1o} \preccurlyeq \forall \overline{a : \sigma}.\ \mathbf{fn}(\mathrm{T}_{2i}; \overline{\tau_2}) \to \tau_{2o}/\mathrm{T}_{2o}}\textsc{S-fun}$$

Fig. 9. Context Inclusion & Subtyping of λ_{LR}.

To see these judgments in action, we will go through parts of the typing derivations of two examples from §2.2: `decr` and `ref_join`. Fig. 7 presents the encoding of both examples in λ_{LR}.

Example 1 The translated version of `decr`, together with some annotations describing the contexts at each step, is shown on the left of fig. 7. The Rust's operators greater than (>) and subtraction (-) are modeled respectively as the predefined functions `gt` and `sub`, with the following types:

$$\mathtt{gt} : \forall (a_1, a_2 : \mathbf{int}).\ \mathbf{fn}(\mathbf{int}[a_1], \mathbf{int}[a_2]) \to \mathbf{bool}[a_1 > a_2]$$
$$\mathtt{sub} : \forall (a_1, a_2 : \mathbf{int}).\ \mathbf{fn}(\mathbf{int}[a_1], \mathbf{int}[a_2]) \to \mathbf{int}[a_1 - a_2]$$

Type-checking of `decr` begins by applying T-fun to check the function as $\mathbf{fn}(\&_{\mathbf{mut}}\ \mathbf{nat}) \to \lightning$. Consequently, x is assigned type $\&_{\mathbf{mut}}\mathbf{nat}$ in the initial value context inside the function body. The context also contains bindings for the recursive call f and the predefined functions `gt` and `sub`.

Next, since x is a reference, T-deref is used to give $*x$ type **nat**, which is then assigned to y in the value context (by rule T-let). Remember that **nat** abbreviates $\{b.\ \mathbf{int}[b] \mid b \geq 0\}$, so the next instruction *unpacks* the existential with a fresh variable a_y, extending the refinement context with $a_y : \mathbf{int}, a_y \geq 0$ and updating the type of y to $\mathbf{int}[a_y]$ (rule T-unpack).

In the call to `gt`, a_y and 0 are used to instantiate the refinement parameters. The rule T-call, first checks that they have the correct sorts (the well-sorted judgment is defined in the technical appendix Lehmann et al. [2023b]). In this case, they both have sort **int** matching the sorts of a_1 and a_2 declared in the function signature. Given these refinement arguments, rule T-call defines the substitution $\theta = [a_y/a_1][0/a_2]$ to check subtyping for the arguments (rule T-int types integers as

$$
\begin{array}{rrcl}
\textbf{\textit{Values}} & v & ::= & \cdots \mid \mathbf{Vec}_\tau\textbf{::new} \mid \mathbf{Vec}_\tau\textbf{::push} \mid \mathbf{Vec}_\tau\textbf{::index\_mut} \mid \mathbf{vec}_\tau(n, \mathrm{ptr}(\ell, t)) \\
\textbf{\textit{Base Types}} & B & ::= & \cdots \mid \mathbf{Vec}_\tau
\end{array}
$$

$$
\begin{aligned}
\mathbf{Vec}_\tau\textbf{::new} &: \mathbf{fn}() \to \mathbf{Vec}_\tau[0] \\
\mathbf{Vec}_\tau\textbf{::push} &: \forall n\!:\!\mathbf{int}, \rho\!:\!\mathbf{loc}.\ \mathbf{fn}(\rho \mapsto \mathbf{Vec}_\tau[n]; \mathbf{ptr}(\rho), \tau) \to \lightning / \rho \mapsto \mathbf{Vec}_\tau[n+1] \\
\mathbf{Vec}_\tau\textbf{::index\_mut} &: \forall a\!:\!\mathbf{int}.\ \mathbf{fn}(\&_{\mathbf{mut}}\mathbf{Vec}_\tau[a], \{b.\ \mathbf{int}[b] \mid 0 \le b < a\}) \to \&_{\mathbf{mut}}\tau
\end{aligned}
$$

Fig. 10. Extension of λ_{LR} with Vectors.

singletons):

$$(1)\ \Delta_3 \vdash \mathbf{int}[a_y] \preccurlyeq \theta \cdot \mathbf{int}[a_1] \qquad (2)\ \Delta_3 \vdash \mathbf{int}[0] \preccurlyeq \theta \cdot \mathbf{int}[a_2]$$

After applying the substitution, the types match exactly and subtyping is trivially satisfied. Note that the rule T-CALL also needs to check inclusion for the function's input location context (allowing framing). In this case, since `gt` has empty location contexts the requirement is satisfied trivially.

Applying the substitution to the return type of `gt` gives $\mathbf{bool}[a_y > 0]$, which is used as the condition in the if statement. So, rule T-IF checks the then branch in a refinement context extended with the assumption $a_y > 0$. The goal is to prove that the assignment to x is safe in this context. First, by rule T-CALL the result of calling `sub` has type $\mathbf{int}[a_y - 1]$. Then, since x is a reference, the rule T-ASS is used to check the assignment generating the following subtyping constraint:

$$a_y\!:\!\mathbf{int}, a_y \ge 0, a_y > 0 \vdash \mathbf{int}[a_y - 1] \preccurlyeq \{b.\ \mathbf{int}[b] \mid b \ge 0\}$$

Subtyping, via rule S-EX, reduces the above to the following validity query

$$a_y\!:\!\mathbf{int}, a_y \ge 0, a_y > 0 \models a_y - 1 \ge 0$$

which is decided valid in the theory of linear arithmetic. Thus, type-checking of `decr` succeeds.

Example 2 The λ_{LR} version of `ref_join` is shown on the right of fig. 7. The initialization of x (*resp.* y) is translated into λ_{LR} as an allocation followed by an assignment. Therefore, first the rule T-NEW is used to type the allocation. This has three effects: (1) it extends the refinement context with a fresh location ρ_x, (2) it binds x as a strong pointer $\mathbf{ptr}(\rho_x)$, and (3) it marks the new location as uninitialized $\rho_x \mapsto \lightning$. This new location is local, in that the output type and location context of the rule cannot refer to it, which is imposed by well-formedness premises. (Well-formedness is checked *w.r.t.* binders in Δ.) Then, since x is a strong pointer, the rule T-ASS-STRG types the assignment and strongly updates the type of ρ_x to $\mathbf{int}[1]$. The initialization of y proceeds analogously.

Next, to type $\&\mathbf{mut}\ x$ (*resp.* $\&\mathbf{mut}\ y$) the rule T-BSMUT is used and "picks" **nat** as the bound in the premise (via inference as explained in §4.2). This choice has the effect of weakening the type associated to ρ_x (*resp.* ρ_y). At this point, the two branches have the following location contexts:

$$(\text{then})\ \rho_x \mapsto \mathbf{nat}, \rho_y \mapsto \mathbf{int}[2] \qquad (\text{else})\ \rho_x \mapsto \mathbf{int}[1], \rho_y \mapsto \mathbf{nat}.$$

Thus, the rule T-SUB weakens each context to obtain $\rho_x \mapsto \mathbf{nat}, \rho_y \mapsto \mathbf{nat}$ as the join. Finally, after the call to `decr`, the rule T-DEREF-STRG types $*x$ as **nat** which matches the declared return type.

3.3 Extension of λ_{LR} with Vectors.

In §2.3 we presented an API for refined vectors indexed by their size. In this section we show how λ_{LR} is extended with a similar API. We treat vectors as a primitive, *i.e.,* with dedicated typing and operational rules (§3.4). This differs from RUST, where vectors are not a primitive but rather implemented using `unsafe` operations which are properly "encapsulated" [Jung et al. 2017]. In our setting, encapsulated means that if programmers use the exposed vector API but otherwise avoid `unsafe` operations themselves, then their programs should not exhibit unsafe/undefined behavior.

To encapsulate the `unsafe` operations, the implementation of vectors in Rust contains run-time checks to ensure vectors are never accessed with invalid indices. Our extension of λ_{LR} with vectors removes these checks and illustrates how (in principle) `unsafe` operations can also be encapsulated under a refined API with the same safety guarantees.

Fig. 10 summarizes how the system is extended with vectors. Base types are extended with $\mathbf{Vec}_\tau$, *i.e.,* vectors of elements of type τ (which can be indexed by their size). Values are extended with functions on vectors, which are given types mirroring the API described in §2.3. Finally, the value $\mathbf{vec}_\tau(n, \text{ptr}(\ell, t))$, represents a vector that points to a block of memory starting at ℓ that holds n (contiguous) elements of type τ. This value is not part of the surface syntax, and as such does not have a top-level type, but shows up at run-time as part of the operational rules for vectors.

3.4 Soundness of λ_{LR}.

We ensure soundness of λ_{LR}—extended with vectors—by proving standard progress and preservation theorems. For space restrictions, we only give a high-level description of the soundness theorem. The detailed proofs and the full definition of our call-by-value small-step operational semantics can be found in the technical appendix [Lehmann et al. 2023b].

The operational semantics follows the *Stacked Borrows* aliasing discipline [Jung et al. 2019]. In Stacked Borrows, pointers $\text{ptr}(\ell, t)$ are tagged and for each location, additional state is used to track existing pointers to the location. The extra state is used to detect violations of Rust's borrowing rules at run-time. We define our operational semantics to return an error if any such violation is detected. Concretely, given a heap h mapping locations to values and a stacked borrows state ς, an expression e can take an evaluation step $\langle h, \varsigma, e\rangle \rightsquigarrow \langle h_o, \varsigma_o, e_o\rangle$ or return an aliasing violation error $\langle h, \varsigma, e\rangle \rightsquigarrow \mathsf{ERR}$. We say that an evaluation is *well-borrowed* when it does not return an error.

To relate the run-time state with the static type system, we define a *dynamic environment* that maps value pointers to pointer types (either a strong pointer or a reference). Then, we extend the typing judgment with an extra context Σ and give pointers $\text{ptr}(\ell, t)$ type $\Sigma(\ell, t)$. Finally, we define a well-typed state relation $\mathrm{T}; \Sigma \vdash \langle h, \varsigma\rangle$, that intuitively states that—if the stacked borrows rules are followed—it is safe to read from a pointer $\text{ptr}(\ell, t)$ at type $\Sigma(\ell, t)$.

With these definitions in place, we proved the following soundness statement:

Theorem 3.1 (Soundness). *If* $\emptyset; \emptyset; T_i; \Sigma_i \vdash e_i : \tau \dashv T$, $T_i; \Sigma_i \vdash \langle h_i, \varsigma_i\rangle$, *and* $\langle h_i, \varsigma_i, e_i\rangle \rightsquigarrow^\star \langle h, \varsigma, e\rangle$, *then one of the following holds*

(1) $\langle h_o, \varsigma_o, e_i\rangle \rightsquigarrow \mathsf{ERR}$, *or*
(2) e *is a value and there exist* T_o *and* $\Sigma_o \supseteq \Sigma_i$ *such that* $\emptyset; \emptyset; T_o; \Sigma_o \vdash e : \tau \dashv T$, *or*
(3) there exists T_o, $\Sigma_o \supseteq \Sigma_i$, h_o, ς_o, *and* e_o *such that* $\langle h, \varsigma, e\rangle \rightsquigarrow \langle h_o, \varsigma_o, e_o\rangle$, $T_o; \Sigma_o \vdash \langle h_o, \varsigma_o\rangle$, *and* $\emptyset; \emptyset; T_o; \Sigma_o \vdash e_o : \tau \dashv T$.

That is, *well-borrowed evaluations of well-typed programs do not get stuck.* This implies, for example, that vectors are always accessed with valid indices.

4 ALGORITHMIC VERIFICATION

Flux implements the type checking rules presented in §3 as a Rust compiler plugin, adding an extra analysis step to the compiler pipeline. As a plugin, Flux operates on programs that have already been analyzed by the compiler. This has two major benefits. First, the compiler's intermediate representations are elaborated with inferred type information which is used by our analysis. Second, we can assume programs satisfy Rust's borrowing rules, which our analysis relies on.

Concretely, Flux performs its analysis on the compiler's Mid-level Intermediate Representation (MIR). The MIR is a control-flow graph (CFG) representation, unlike our core calculus, which relies on recursive functions to represent complex control-flow constructs. However, both representations are easy to relate via the correspondence defined by Appel [2007].

Still, there are three key challenges to address in bridging the gap between the formalism presented in §3 and our implementation. First (§4.1), the syntax of λ_{LR} has explicit refinement annotations that do not appear in Rust's MIR. Second (§4.2), some judgments in λ_{LR} have rules (*e.g.,* T-fun, T-bsmut, and T-sub) with a non-deterministic choice of types that the implementation needs to infer. Finally (§4.3), Flux supports polymorphic types which are crucial for ergonomic specification and verification, but require instantiating type parameters with refinement types.

4.1 Refinement Annotations

Flux, following the essence of refinement typing, does not modify the syntax of Rust programs, but allows refined function signatures. Thus, users must declare refined signatures for top-level functions (using the syntax described in §2), but the placement of **unpack** instructions and the instantiation of refinement parameters at function calls are automatically inferred by Flux.

Flux places **unpack** instructions implicitly and *on-the-fly*, by eagerly generating a fresh refinement variable as soon as an existential type enters the value context. As an example, recall the translated version of `decr` (§3.2). As soon as $y : \{b.\ \textbf{int}[b] \mid 0 \geq b\}$ is introduced in the value context, Flux places an implicit $\textbf{unpack}(y, a_y)$ with a fresh refinement variable a_y.

The instantiation of refinement parameters is performed by syntax-directed unification during subtyping [Economou et al. 2022]. For example, in the function `decr`, the comparison `y > 0` is encoded in λ_{LR} as $\textbf{call}\ \texttt{gt}[a_y, 0](y, 0)$, where a_y and 0 instantiate the parameters a_1 and a_2 in the signature of `gt`:

$$\forall(a_1, a_2 : \textbf{int}).\ \textbf{fn}(\textbf{int}[a_1], \textbf{int}[a_2]) \rightarrow \textbf{bool}[a_1 > a_2]$$

In the implementation, the instantiation for a_1 and a_2 needs to be inferred. To do so, when the types of the actual arguments $\textbf{int}[a_y]$ and $\textbf{int}[0]$ are checked for subtyping against the type of the formals $\textbf{int}[a_1]$ and $\textbf{int}[a_2]$, we unify their indices and instantiate a_1 to a_y and a_2 to 0.

To guarantee that unification always succeeds, refinement parameters must be restricted to appear in certain positions within types. The exact restrictions are best described by Economou et al. [2022], but the intuition is that a refinement parameter must be used at least once as an index in argument position. Since the surface syntax does not expose explicit quantification, but rather uses the `@n` syntax to declare refinement parameters, we can guarantee that signatures satisfy the restrictions by restricting the positions where `@n` can be used. For example, Flux will reject the signature `fn() -> i32[@n]`, because the parameter `n` is declared in return position. More notably, parameters cannot be declared inside a type argument of a polymorphic type constructor. This restriction can be circumvented by declaring extra parameters. For instance, the function `normalize_centers` in fig. 4 binds `n` by taking an extra `usize[@n]` as the first argument, because Flux would reject the signature if declared as:

```
fn(&mut RVec<RVec<f32>[@n]>[@k], &RVec<f32>[k]))
```

As a limitation of the current implementation, `n` must be attached to a computational parameter, but this is in principle a ghost parameter.

4.2 Refinement Inference

Several rules of fig. 8 have cases where types are inferred. For example, T-fun guesses the type of the function, T-bsmut guesses the type of the resulting mutable reference, and T-sub guesses weakened types, allowing unification at join points and function calls. Flux's inference proceeds in *three phases.* To illustrate these phases, let us see how types can be inferred at the join point after the **if** statement in the `ref_join` function (§3.2). In summary, the application of the rules requires inferring three types τ_1, τ_2, and τ satisfying the following requirements:

$$(1)\ a \vdash \mathbf{int}[1] \preccurlyeq \tau_1 \quad (2)\ \neg a \vdash \mathbf{int}[2] \preccurlyeq \tau_2 \quad (3)\ \emptyset \vdash \&_{\mathbf{mut}}\ \tau_1 \preccurlyeq \tau \quad (4)\ \emptyset \vdash \&_{\mathbf{mut}}\ \tau_2 \preccurlyeq \tau$$

In the then branch, the borrow of x weakens its type from $\mathbf{int}[1]$ to τ_1 (by T-BSMUT) leading to *(1)*. Similarly, in the else branch borrowing y leads to *(2)*. Finally, in the assignment to r, the reference type in each branch must be unified to a common type τ, leading to *(3)* and *(4)*.

Phase 1: Shape Inference Flux begins by inferring the *shape* of the types to the most general one that can satisfy the subtyping requirements. In our example, to satisfy *(1)* we know that either S-IDX or S-EX must apply. Thus, τ_1 must be either an indexed type or an existential. We note that, without changing the typability of the program, we can always choose τ_1 to be an existential, because for any indexed type $B[n]$ we have $B[n] \preceq \{b.\ B[b] \mid b = n\}$ and $\{b.\ B[b] \mid b = n\} \preceq B[n]$. Therefore, Flux determines τ_1 to have the shape:

$$\tau_1 \doteq \{b.\ \mathbf{int}[b] \mid \kappa_1(b)\}$$

Crucially, this type contains a *refinement predicate* κ_1, *i.e.,* an *unknown* predicate whose value will be decided in the next phase. Similarly, Flux determines the shapes of τ_2 and τ to be:

$$\tau_2 \doteq \{b.\ \mathbf{int}[b] \mid \kappa_2(b)\} \quad \tau \doteq \&_{\mathbf{mut}}\ \{b.\ \mathbf{int}[b] \mid \kappa(b)\}$$

Phase 2: Constraint Generation Next, Flux uses the type checking rules to generate a *verification condition* (VC) that constrains the unknown predicates. For our example, it yields the below VC:

$$\begin{aligned}
&(1)\ a \Rightarrow \kappa_1(1)\ \wedge \\
&(2)\ \neg a \Rightarrow \kappa_2(2)\ \wedge \\
&(3)\ \kappa_1(b) \Rightarrow \kappa(b) \wedge \kappa(b) \Rightarrow \kappa_1(b)\ \wedge \\
&(4)\ \kappa_2(b) \Rightarrow \kappa(b) \wedge \kappa(b) \Rightarrow \kappa_2(b) \\
&(5)\ \kappa(b) \Rightarrow b \geq 0 \wedge b \geq 0 \Rightarrow \kappa(b) \\
&(6)\ \kappa(b) \Rightarrow b \geq 0
\end{aligned}$$

Here, the conjuncts *(1)* to *(4)* correspond to the subtyping requirements generated in the **if** statement, while *(5)* and *(6)* correspond to the constraints generated by the call to `decr` and the return type respectively.

Phase 3: Liquid Inference Finally, Flux uses the Liquid Fixpoint[3] horn constraint solver to synthesize a solution for the unknown predicates using predicate abstraction. More concretely, it finds a solution for the k-predicates over an *abstract domain* of formulas generated by conjunctions of predefined atomic predicates [Cosman and Jhala 2017]. In our example, Liquid Fixpoint finds the solution $\kappa(b), \kappa_1(b), \kappa_2(b) := b \geq 0$, which satisfies the original subtyping requirements. In general, the unknown κ predicates are Horn variables that may have *multiple* arguments, allowing liquid inference to track dependencies between multiple program variables, thereby enabling Flux to automatically synthesize loop invariants.

4.3 Polymorphic Instantiation

As described in §2, Flux exploits polymorphism to infer invariants over elements of polymorphic type constructors. To achieve this, Flux instantiates type parameters with existentials containing unknown predicates. For instance, consider the function below that creates a single element vector.

[3] https://github.com/ucsd-progsys/liquid-fixpoint

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 **(% LOC)** is the ratio of loop-invariant lines to RUST source code, and **Time (s)** is the time in seconds required to verify the code (trusted code does not have time).

	FLUX			PRUSTI				
	LOC	**Spec**	**Time (s)**	**LOC**	**Spec**	**Annot**	**(% LOC)**	**Time (s)**
Library								
RVec	41	20	-	45	29	-	-	-
RMat	22	6	0.21	33	15	-	-	-
Total	63	26	0.21	78	44	-	-	-
Benchmark								
bsearch	25	1	0.18	25	0	1	4%	3.25
dotprod	12	1	0.14	12	1	1	8%	2.75
fft	162	7	0.70	188	22	24	12%	166.76
heapsort	37	2	0.22	37	5	9	24%	8.25
simplex	118	8	0.45	125	25	8	6%	12.19
kmeans	85	8	0.43	87	37	10	11%	13.41
kmp	48	2	0.51	49	4	7	14%	10.23
Total	487	29	2.63	423	94	60	14%	217.91
Case Study								
WaVe	5585	318	16	5585	1001	47	0.8%	2040

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

The comments show the type of vec after each statement. In the call to new, FLUX needs to instantiate the parameter T in the return type RVec<T>[0]. We extract from the RUST compiler that T needs to be an i32 but its refinement is unknown. Thus, FLUX instantiates T with the template i32$\{\nu : \kappa_1(\nu)\}$ where κ_1 is a fresh unknown predicate. Similarly, the call to push generates the template i32$\{\nu : \kappa_2(\nu)\}$. Type-checking the program with these templates generates the (Horn) VC $(\kappa_1(\nu) \Rightarrow \kappa_2(\nu)) \wedge (\nu = 42 \Rightarrow \kappa_2(\nu)) \wedge (\kappa_2(\nu) \Rightarrow \nu > 0)$. The first two conjuncts correspond to subtyping for the two arguments to push; the third relates the type of vec to the output type. Using liquid inference, FLUX solves $\kappa_1(\nu) := \nu > 0$ and $\kappa_2(\nu) := \nu > 0$ which is strong enough to check the above verification condition is valid, and hence, verify the type of make_vec.

5 EVALUATION

Next, we present an empirical evaluation of the benefits of FLUX's refinement type-based, lightweight verification compared 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 *deep* verification, *i.e.,* it allows users to verify various forms of functional correctness properties (*e.g.,* sorted-ness) not expressible in FLUX. However, we show that for many common and important use-cases, FLUX's type-based lightweight verification is more attractive.

In particular, our evaluation focuses on three dimensions for comparison: do types (§5.2) enable *compact* specifications? (§5.3) require *fewer* annotations? (§5.4) facilitate *faster* verification?

5.1 Benchmarks

We compare Flux and Prusti on two sets of benchmarks: a set of vector-manipulating programs from the literature and a larger case study porting WaVe: a Rust-based Web-Assembly sandboxing runtime [Johnson et al. 2023].

Case Study: Vector Bounds Checking Our first set of benchmarks is a set of vector-manipulating programs drawn from the literature [Rondon et al. 2008], which implement loop-heavy algorithms over the `RVec` library discussed in §2.3. Some benchmarks use `RMat`, a refined 2-dimensional matrix indexed by the number of rows and columns, which was implemented on top of `RVec` as a vector of vectors. 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 verified the code in Flux and then replicated it as closely as possible in Prusti.

Case Study: Verified Sandboxing in WaVe Our second set of benchmarks is from a Web-Assembly sandboxing library previously written in Rust and already verified with Prusti [Johnson et al. 2023]. This case study evaluates whether Flux's refinement types are expressive enough to capture real-world security requirements, while still offering advantages in terms of annotation and verification overhead. In brief, the security properties include checking that (1) *vectors* and *slices* are accessed within their bounds, (2) *memory accesses* granted by the sandbox stay within the sandbox's memory region and (3) *symbolic links* in filepath components are fully resolved to point within the sandbox. We refer the reader to [Johnson et al. 2023] for more details. For our case study, we ported all the Prusti specifications in WaVe to Flux. Crucially, we were able to use Flux's refined `struct` mechanism to compactly capture all the secure sandboxing specifications as refinement types, thereby using plain Rust typing and polymorphism to entirely eliminate quantifiers from the specifications.

Setup We ran all the experiments on a laptop running Fedora 36 with 32GB of memory and a 12th Gen Intel(R) Core(TM) i7-1280P CPU. We used the following versions of the software required to run Prusti: (1) Prusti commit `673a095d`, (2) Z3 v4.8.6, and (3) openjdk-17.0.4.1. 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. 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. We emphasize that the main takeaway with the case study row is just that Flux's refinement types are expressive enough to specify and efficiently verify the security properties for a complex sandbox [Johnson et al. 2023]. The reduction in specification size (and hence, likely verification time) is largely because we were able to encapsulate the requirements as refined structs *e.g.,* that representing the "Virtual Machine Context" (`VmCtx`) which then let us replace a swathe of Prusti's `requires` and `ensures` clauses simply with the Rust type signature. Finally, it is likely that if the Prusti developers rewrote the specifications, they could exploit their knowledge of what yields the

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

// Prusti Specification ----------------------------------------------------------
#[requires(idx < self.len())]                                              // (1)
#[ensures(self.len() == old(self.len()))]                                  // (2)
#[ensures(forall(|i: usize| (i < self.len() && i != idx) ==>               // (3)
                                 self.lookup(i) == old(self.lookup(i))))]
#[ensures(self.lookup(idx) == value)]                                      // (4)

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

Fig. 11. The specifications for `RVec::store` in Rust, Prusti and Flux,

most efficiently solved VCs to shrink specification size—*e.g.,* by using the type invariant mechanism [Astrauskas et al. 2022] to compactly specify the requirements on `VmCtx`—and hence, verification time.

5.2 Compact API Specifications

The column **Spec** in table 1 shows 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. 11. (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 in Prusti `requires` the index to be within bounds (condition 1) and `ensures` that the vector has the same length after the function returns (condition 2) and all the elements in the vector remain unchanged except for the one being updated which instead gets the new value (conditions 3 and 4).

Prusti's specification is strictly more expressive than Flux's specification. For instance, Prusti's specification can be used to verify *relational* properties between elements (*e.g.,* sortedness). In contrast, Flux's signature, which relies on polymorphism, can only express *unary* predicates that must be true for all elements. However, quantifying over the elements is necessary to verify `kmeans`, `kmp`, and `wave`, as these benchmarks store pointers and array indices *within* containers, and hence require tracking invariants of those indices using quantified contracts. For the rest of the benchmarks, a weaker specification that only tracks the vector's length (*i.e.,* conditions 1 and 2) is sufficient.

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: up to 24% (average 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. 11) 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, requiring zero annotations.

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.

5.4 Faster Verification

The columns **Time (s)** in table 1 show times taken by Flux and Prusti for each benchmark. To be fair to Prusti, we only use the full specification for `store` when necessary and default to the weaker one otherwise (without conditions 3 and 4). Prusti consistently takes at least one order of magnitude longer to verify each benchmark, taking close to 3 minutes to verify `fft`, verified by Flux in 0.7 sec. Note that Flux is faster despite spending time to synthesize loop invariants, unlike Prusti, where this information is furnished by the user. We speculate that there are at least two different reasons for the gap. First, with Prusti's VCs, the SMT solver must *instantiate* and *check* quantified loop invariants which are known to cause performance issues in SMT solvers [Leino and Pit-Claudel 2016]. Second, verification with Prusti implicitly constructs proofs for memory safety (via reduction to the Viper IR), which is an overhead Flux avoids by relying upon Rust's guarantees.

While this is advantageous for our benchmarks, PRUSTI's approach allows it to be used to verify `unsafe` code, which presently impossible with FLUX. Further experimentation with PRUSTI may shed more light on the gap and yield optimizations that bring the verification times closer.

6 RELATED WORK

RUST formal semantics The Stacked Borrows [Jung et al. 2019] aliasing discipline proposes an operational semantics for RUST with the intention of defining *undefined behavior* when memory accesses through references and raw pointers are combined. Our formalization (§3), uses stacked borrows to characterize the requirements on memory accesses that FLUX relies on.

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. To achieve this they define a semantic interpretation of RUST ownership types in IRIS [Jung et al. 2018] and prove that a library using unsafe operations satisfies the predicates of its interface semantic interpretation. It would be interesting to extend RUSTBELT with refinement types and use the same semantic approach to prove libraries using unsafe operations can be encapsulated 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, it is based on an interpretation of lifetimes as *provenance sets*, and resembles the prototype borrow checker implementation Polonius [Matsakis 2018].

Refinement types and imperative code Refinement types were originally developed for the verification of functional programs with ML style references [Freeman and Pfenning 1991; Xi and Pfenning 1999b]. Many of these early ideas were summarized in the Applied Type System framework (ATS) [Xi 2004] which further supported pointer manipulation via a notion of *stateful views* that required manually provided proofs to track ownership [Zhu and Xi 2005].

The idea of synthesizing refinements via liquid type inference was also introduced for an ML like language [Rondon et al. 2008] and latter extended to 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. Like ATS, this line of work focuses on the manipulation of 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.

Asynchronous liquid separation types Kloos et al. [2015] describes a type system that combines refinements with a concurrent separation logic to verify asynchronous Ocaml programs with mutable state. More recently, Sammler et al. [2021] proposed a type system that combines ownership and refinement types to provide automated verification for C programs. Their 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 RUST with refinement types resembles REFINEDC, but their model of ownership is different from RUST's references and requires the manual annotation of loops to track ownership.

RUST verification tools Several program logic based tools exist for the verification of heavyweight functional correctness properties of RUST programs. PRUSTI encodes programs into VIPER [Müller et al. 2016]; RUSTHORN [Matsushita et al. 2021] generates constrained Horn clauses [Bjørner et al. 2015]; and CREUSOT [Denis et al. 2022] extracts programs into WhyML [Filliâtre and Paskevich 2013]. Ullrich [2016] defines an encoding of safe RUST into a functional program, which can be interactively verified in Lean [de 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. Ho and Protzenko [2022] extend the translation to support mutation via backward functions, which, like *lenses*, update the heap post-mutation. All these tools leverage Rust's ownership types to abstract the low level details of reasoning about aliasing and to provide a specification language in a program logic. As discussed in §5, using a program logic comes at the cost of complex user-specified universally quantified invariants. In contrast Flux aims to make lightweight verification automatic and ergonomic by restricting specifications so that that the type system itself becomes a syntax-directed decision procedure for universally quantified assertions, thereby enabling automatic (quantifier-free) invariant inference, and eliminating programmer overhead. *Bounded* verification of Rust programs has also been done 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, use 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 correctness properties which are currently out of Flux's reach. In future, it would be interesting to see how to recoup such expressiveness, perhaps by incorporating techniques like reflection [Vazou et al. 2018], that has proven effective in the purely functional setting.

8 DATA AVAILABILITY STATEMENT

The source code of Flux is publicly available at https://github.com/flux-rs/flux. Additionally, a snapshot of the software used for the evaluation in §5, together with instructions on how to replicate the results can be found in the accompanying artifact [Lehmann et al. 2023a].

ACKNOWLEDGMENTS

We thank the reviewers who evaluated the artifact for sharing their experience using Flux, our shepherd Peter-Michael Osera and the anonymous referees for their excellent suggestions for improving the paper, and Gilles Barthe who helped us flesh out the design of Flux in its early stages. This work was supported by the NSF grants CNS-2120642, CNS-2155235, CCF-1918573, CCF-1911213, the Horizon Europe ERC Starting Grant CRETE (GA: 101039196), the US Office of Naval Research HACKCRYPT (Ref. N00014-19-1-2292), and generous gifts from Microsoft Research.

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. https://doi.org/10.1017/CBO9780511609619

Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods*, Jyotirmoy V.

Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer International Publishing, Cham, 88–108. https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147 (oct 2019), 30 pages. https://doi.org/10.1145/3360573

Alexander Bakst and Ranjit Jhala. 2016. Predicate Abstraction for Linked Data Structures. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 65–84. https://doi.org/10.1007/978-3-662-49122-5_3

Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. *SIGOPS Oper. Syst. Rev.* 51, 1, 94–99. https://doi.org/10.1145/3139645.3139660

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. https://doi.org/10.1007/978-3-319-23534-9_2

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* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

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. *Proc. ACM Program. Lang.* 1, ICFP, Article 26 (aug 2017), 27 pages. https://doi.org/10.1145/3110270

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering*, Adrian Riesco and Min Zhang (Eds.). Springer International Publishing, Cham, 90–105. https://doi.org/10.1007/978-3-031-17244-1_6

Dimitrios J Economou, Neel Krishnaswami, and Jana Dunfield. 2022. Focusing on Liquid Refinement Typing. *arXiv preprint arXiv:2209.13000* (2022).

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468

Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (aug 2022), 31 pages. https://doi.org/10.1145/3547647

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

E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown. 2023. WaVe: A Verifiably Secure WebAssembly Sandboxing Runtime. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1986–2001. https://doi.org/10.1109/SP46215.2023.00114

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (dec 2019), 32 pages. https://doi.org/10.1145/3371109

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. 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), e20. https://doi.org/10.1017/S0956796818000151

Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. 2012. Deterministic Parallelism via Liquid Effects. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 45–54. https://doi.org/10.1145/2254064.2254071

Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. 2015. Asynchronous Liquid Separation Types. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*, John Tang

Boyland (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 396–420. https://doi.org/10.4230/LIPIcs.ECOOP.2015.396

Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023a. *Flux: Liquid Types for Rust - Artifact.* https://doi.org/10.5281/zenodo.7682468

Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023b. Flux: Liquid Types for Rust - Technical Appendix. https://github.com/flux-rs/pldi23-artifact

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, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). 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)*. 108–114. https://doi.org/10.1109/INDIN.2018.8471992

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

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.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

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, Vol. 1445)*, Eric Jul (Ed.). Springer, 158–185. https://doi.org/10.1007/BFb0054091

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–67. https://doi.org/10.1007/978-3-540-28644-8_4

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, USA, 55–74.

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-Level Liquid Types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 131–144. https://doi.org/10.1145/1706299.1706316

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

J. Rushby, S. Owre, and N. Shankar. 1998. Subtypes for specifications: predicate subtyping in PVS. *IEEE Transactions on Software Engineering* 24, 9 (1998), 709–720. https://doi.org/10.1109/32.713327

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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

Frederick Smith, David Walker, and Greg Morrisett. 2000. Alias Types. In *Programming Languages and Systems*, Gert Smolka (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 366–381.

John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 684–714. https://doi.org/10.1007/978-3-030-44914-8_25

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. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 321–330. https://doi.org/10.1145/3510457.3513031

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).

Hongwei Xi. 2004. Applied type system. In *Types for Proofs and Programs: International Workshop, TYPES 2003, Torino, Italy, April 30-May 4, 2003, Revised Selected Papers*. Springer, 394–408. https://doi.org/10.1007/978-3-540-24849-1_25

Hongwei Xi and Frank Pfenning. 1999a. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

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* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

Dengping Zhu and Hongwei Xi. 2005. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages* (Long Beach, CA) *(PADL'05)*. Springer-Verlag, Berlin, Heidelberg, 83–97. https://doi.org/10.1007/978-3-540-30557-6_8

Received 2022-11-10; accepted 2023-03-31