Karlsruhe Institute of Technology

KASTEL

Exclusivity through Fractional Permissions for Java

Bachelor's Thesis by

David Kiefer

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner: Prof. Bernhard Beckert
First advisor: Florian Lanzinger, M.Sc.

10. November 2025 – 10. March 2026

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

Property types are a way to specify program behavior through a type system. They are refinement types that reason about user-defined properties. When verifying, some properties can be discharged using a type checker that provides good usability. The remaining properties can then be checked by a more powerful tool for deductive verification to eliminate most false positives.
To be able to verify mutable data structures, property types need a system that ensures that only exclusive references can be mutated. The Kukicha framework for property types provides a Uniqueness Checker that ensures this property. The type system that it enforces, however, enforces rather conservative rules, leading to many false positives.
In this thesis, we develop a type system that upholds this invariant while keeping a high degree of expressiveness. We formalize its type rules on a minimal imperative language and provide a formal proof that it ensures the exclusivity requirements. We present *borro*$\mathbb{Q}$, an implementation of this type system using the Checker Framework. For this, we adapt the type rules to Java, balancing adherence to the proven formal specification and usability.

Contents

1 Introduction

Precise aliasing information is useful in many areas of static analysis and compiler design. We focus on the specific aliasing property that ensures that references can only be mutated if there is no other reference to the same object. We call these references *unique* or *exclusive*. For concurrent programs, this condition can be used to guarantee that parallel execution is deterministic [10]. It can also enable significant compiler optimizations by allowing the compiler to use aliasing guarantees that the property provides [14], [25]. Rust [20] uses a similar system through its *borrow checker* along with an *ownership* concept for memory management [21]. We mainly motivate our thesis through the lens of *formal verification* and specifically *property types*.

The two most prominent approaches for verifying properties of programs are *type checking* and *deductive verification*. Both techniques, however, suffer from different limitations.

Type checking is often directly integrated into a language and thus offers good usability for the programmer. Languages with static type systems have gained widespread adoption, including some with strong type systems like Rust [20]. However, type systems are often limited in the properties they can verify. Semantic meaning of a program can typically not be expressed through type checking. Even if the type system is theoretically expressive enough to state these properties, as is the case in TypeScript which allows arbitrary execution in the type system, the type checker's execution is bounded. Additionally, the type checker's data flow and control flow analyses might be limited, as shown in the following example which can never take the branch that returns a `string` but still does not type-check:

```typescript
function id(x: number): number {
    if (true) {
        return x
    } else {
        return "hi"
    }
}
```

TypeScript

Deductive verification, on the other hand, is very expressive and can use techniques like data flow analysis and symbolic execution to achieve high precision by producing less false positives. Tools for deductive verification are most often external and thus not well integrated into the language. One example of this is the Java Modeling Language (JML) [8] that allows the user to specify properties of Java [11] programs with comments source

code. This specification then has to be checked by an external (interactive) tool like KeY [1]. This results in deductive verification not being very ergonomic.
With *Kukicha*, F. Lanzinger *et al.* [16] introduce *property types* for Java that combine type systems and deductive verification to create a system that is both easy to use and expressive. They allow to specify properties of a program using type annotations. When verifying the program, the type-checker first discharges any properties it can infer from basic transitions specified in Kukicha. The remaining properties are then translated to JML for further analysis through deductive verification.
J. Bachmeier [2] shows in their thesis that property types can be applied to mutable data structures if we can guarantee that mutation only occurs on unique references. Kukicha contains a uniqueness type system originally developed in J. Bachmeier's thesis that provides this guarantee. This type system, however, is very limited in its expressiveness since it quickly loses its uniqueness guarantees. This thesis thus aims to develop a new type system that provides the necessary invariants while allowing a wider range of correct programs to successfully type-check. We try to give a justification for this requirement in the following.

1.1 Aliasing and Mutability

Java

```java
@MinLength(5) List f() {
  @MinLength(5) List l5 = ...;
  @MinLength(4) List l4;

  l4 = l5;

  l4.removeFirst(); // -> l4: @MinLength(3)

  return l5; // This might not have length 5 anymore
}
```

Listing 1.1: Problematic interaction of property types, mutability, and aliasing.

Mutability becomes problematic for property types if it interacts with aliasing. We demonstrate this in Listing 1.1 which we adapt from J. Bachmeier's thesis [2]. In this example, `l4` and `l5` are aliases of each other. Thus, the mutating method call `removeFirst` also modifies `l5`. If we do not properly track this, we could violate the `@MinLength` annotation on `l5` and return a value that does not conform to the specification.

1.2 Foundations

We base our work on *fractional permissions* which were originally introduced by J. Boyland [4]. Fractional permissions are a system for alias tracking that allows to split a permission into multiple read-only permissions but then later recombine them to regain write access. We explain how these fractional permissions work in Section 3.1.1.
To implement the type system for Java, we use the Checker Framework [19]. The Checker Framework is a framework for *pluggable types*. Pluggable type systems extend the existing type system of a language by an additional layer of custom types without changing the language semantics. The Checker Framework provides an easy way to specify these custom types using *annotations* and provides ways to implement custom type rules. The most well-known example of a pluggable type system in Java is a nullness type system. With the help of user-specified annotations including `@Nullable` and `@NonNull`, a type checker like the Checker Framework's Nullness Checker can ensure that a Java program does not exhibit any `NullPointerExceptions` at runtime.

1.3 Overview

In this work, we develop a type system that ensures that every mutable access only occurs on a unique reference. For this type system we use a minimal language which we describe in section 2. We explain how this type system works and justify some design decisions in subsection 3.1. Then we provide a formalization of this description through the type rules in subsection 3.2. In section 4, we prove that the type system we developed ensures that only unique references are mutable.
We describe the implementation for our type system in section 5 and evaluate it in section 6 by comparing it with other similar solutions. In section 7 we discuss other approaches to our problem in existing literature.
We conclude our work in section 8 where we discuss advantages and disadvantages of our approach and provide some ideas for future work.

2 Language

For the formalization of our type system, we use an imperative language with a Java-inspired syntax that is defined in Table 2.1. This language is intentionally minimal to reduce the complexity of the type system and proof given in section 3 and section 4, respectively. As parameters behave identically to local variables apart from the way they are declared and first defined, we also mean parameters if we speak of local variables in the following. We introduce the following restrictions in which our language differs from typical imperative languages:

1. `bool` is the only primitive datatype.
2. User-defined structures are the only reference data type.
3. Arguments to function calls must be local variables.
4. Constructors are implicitly defined and take values for each field. They have no body and thus contain no logic.
5. Field accesses are only allowed on local variables.
6. Assignments must have a local variable on the left or right hand side.
7. User-defined structures have exactly two fields: `a` and `b`.
8. Functions have exactly two parameters: `a` and `b`.
9. If-statements with a local variable as the condition are the only control-flow structure apart from function calls.
10. Only user-defined structures can be returned.

These syntactical restrictions do not constrain the expressiveness of our language. The argument/field count as well as the restriction to boolean primitives can be worked around by nested structures. We can get around the restriction of arguments to local variables and the single field accesses by introducing temporary variables. Loops can be transformed to recursive functions. Some of these transformations are shown in Listing 2.1.
The biggest difference to Java is that our language does not have inheritance. We go into more detail on how we apply our type system formalization to Java in section 5 and talk about ways to better support inheritance in subsection 5.4.

```
1  class A {
2    bool x;
3  }
4
5  class B {
6    mutable A a;
7    immutable A b;
8    mutable A c;
9  }
10
11 bool main(immutable B b) {
12   return b.a.x == b.c.x;
13 }
```

(a) Unrestricted code

```
1  class A {
2    bool a;
3    bool b;
4  }
5  class C {
6    immutable A a;
7    mutable A b;
8  }
9  class B {
10   mutable A a;
11   mutable C b;
12 }
13
14 immutable A main(bool a,
   immutable B b) {
15   immutable A ba = b.a;
16   immutable C b_temp = b.b;
17   immutable A bc = b_temp.b;
18   bool bax = ba.a;
19   bool bcx = bc.a;
20
21   bool res;
22   if (bax) {
23     res = bcx;
24   } else {
25     if (bcx) {
26       res = false;
27     } else {
28       res = true;
29     }
30   }
31   immutable A res_ref = new
     A(res, false);
32   return res_ref;
33 }
```

(b) Restricted code in our minimal language. For brevity, variable declaration and first assignment are combined.

Listing 2.1: Example transformation of a small example

Identifier	ident	::=	`[A-Za-z][A-Za-z0-9_\-]*`	
Scope	scope	::=	{ident ∗}	Without *base*
		|	{*base*, ident ∗}	With *base*
Mutability	mutability	::=	`mutable`	
		|	`immutable`	
Program	program	::=	class-decl ∗ fun-decl ∗	
Field type	field-type	::=	mutability ident	Reference type
		|	`bool`	Boolean
Parameter type	param-type	::=	mutability scope ident	Reference type
		|	`bool`	Boolean
Class declaration	class-decl	::=	`class` ident `{`	
			field-type `a;`	
			field-type `b;`	
			`}`	
Function declaration	fun-decl	::=	mutability ident ident `(`	
			param-type ident`,`	
			param-type ident`,`	
			`) {` *s* `}`	

Function Body

Variable	x, y, l, m, n, b	::=	ident	local variable
Boolean Literal	bool-lit	::=	`true`	
		|	`false`	
Statement	*s*	::=	ident l`;`	Variable declaration
		|	`bool` b`;`	Boolean declaration
		|	l `=` `new` ident`(`x`,`y`);`	Reference creation
		|	l `=` ident`(`m`,`n`);`	Method call
		|	l `=` m`;`	Ref-ref assignment
		|	x `=` l`.`f`;`	Var-field assignment
		|	l`.`f `=` y`;`	Field-var assignment
		|	b `=` bool-lit`;`	Bool-Lit reassignment
		|	`return` l`;`	Return
		|	`if (`b`) {` *s* `} else {` *s* `}`	Conditional statement
		|	*s* *s*	Concatenation

Table 2.1: Language syntax

2.1 Operational Semantics

We define the semantics of our language using *states*, represented by a partial function σ, mapping variables and fields to their values. We adapt these semantics from F. Lanzinger *et al.* [16] but reduce the complexity since we do not need labels and the packing system. We specify the syntax using rules for transitions $\sigma \overset{s}{\rightsquigarrow} \sigma'$ (for more complex s, we write $\sigma, s \rightsquigarrow \sigma'$) where s is a statement that transitions the current state σ to σ'.

Universe of values Values in our semantic model are either boolean ($\mathbb{B}$) or of a custom type. We call the set of all values of custom types $\mathfrak{C}$. The set of all values is the universe $\mathfrak{U} = \mathbb{B} \cup \mathfrak{C}$.

Notation We write $f' = f \lhd (a : b)$ for the (partial) function with $f'(a) = b$ and $f'(x) = f(x)$ for all $x \neq a$. For returning a value, we introduce a *marker variable* $retval$ that is assigned by return statements and read by the caller.

Value mapping For every variable and parameter v, σ maps the variable to its value: $\sigma(v) \in \mathfrak{U}$. Additionally, $\sigma(heap)$ is a partial function $\mathfrak{C} \rightharpoonup \{\texttt{a}, \texttt{b}\} \to \mathfrak{U}$ that maps the fields of custom values to their respective values.

Using σ, we define an evaluation function eval_σ:

$$\begin{aligned} \text{eval}_\sigma(\beta) &= \beta && \text{for boolean literal } \beta \\ \text{eval}_\sigma(x) &= \sigma(x) && \text{for a variable } x \\ \text{eval}_\sigma(l.f) &= \sigma(heap)(\text{eval}_\sigma(l))(f) && \text{for a field access} \end{aligned}$$

and the following rules for operational semantics:

$$\frac{}{\sigma, x = e \rightsquigarrow \sigma \lhd (x : \text{eval}_\sigma(e))}\ \textsc{Sem-Assign-Var}$$

$$\frac{}{\sigma, m.f = e \rightsquigarrow \text{store}(\sigma, \text{eval}_\sigma(m), f, \text{eval}_\sigma(e))}\ \textsc{Sem-Assign-Field}$$

$$\frac{}{\sigma, m = \texttt{new } C(e_1, e_2) \rightsquigarrow \text{create}(\sigma, m, C, \text{eval}_\sigma(e_1), \text{eval}_\sigma(e_2))}\ \textsc{Sem-New}$$

$$\frac{\text{body}(f) = s \qquad \sigma_m^{\text{init}} = (\texttt{a} : \text{eval}_\sigma(e_1), \texttt{b} : \text{eval}_\sigma(e_2), heap : \sigma(heap)) \qquad \sigma_m^{\text{init}}, s \rightsquigarrow \sigma'_m}{\sigma, m = f(e_1, e_2) \rightsquigarrow \sigma \lhd (m : \sigma'_m(retval), heap : \sigma'_m(heap))}\ \textsc{Sem-Call}$$

$$\frac{}{\sigma, \texttt{return } m \rightsquigarrow \sigma \lhd (retval : \text{eval}_\sigma(m))}\ \textsc{Sem-Return}$$

$$\frac{\text{eval}_\sigma(b) = \texttt{true} \qquad \sigma, s_1 \rightsquigarrow \sigma'}{\sigma, \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \rightsquigarrow \sigma'}\ \textsc{Sem-If-True}$$

$$\frac{\text{eval}_\sigma(b) = \texttt{false} \qquad \sigma, s_2 \rightsquigarrow \sigma'}{\sigma, \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \rightsquigarrow \sigma'}\ \textsc{Sem-If-False}$$

$$\frac{\sigma, s_1 \rightsquigarrow \sigma' \qquad \sigma', s_2 \rightsquigarrow \sigma''}{\sigma, s_1\texttt{; } s_2 \rightsquigarrow \sigma''}\ \textsc{Sem-Seq}$$

Table 2.2: Operational Semantics Rules

with functions

$$\text{store}(\sigma, l, f, v) = \sigma \lhd (heap : (\sigma(heap) \lhd (l : \sigma(heap)(l) \lhd (f : v))))$$

$$\text{create}(\sigma, l, C, v_1, v_2) = \sigma \lhd (heap : (\sigma(heap) \lhd (l : (\texttt{a} : v_1, \texttt{b} : v_2))))$$

to update fields and create new references.

3 Type System

In this section we describe a type system that ensures our desired property: Mutation is only allowed on exclusive references.

3.1 Design and Rationale

We first give an overview of the design of the type system. We explain the mechanisms that uphold our invariants and justify some design decisions.

3.1.1 Fractional Permissions

Fractional permissions form the basis of our type system. Every variable is assigned a rational *fraction* and an *id*. We denote a variable l that has a fraction p and id id_l assigned as follows:

$$l : [p]_{\text{id\_l}}$$

The *id* identifies the object referenced by this variable and all variables with the same *id* are aliases of each other. The reverse, however, is not true: There may be different *id*s that reference the same object. The *fraction*s then denote what part of the permission of the *id* is allocated to the specific variable.
Newly created ids get assigned a fraction according to their declared mutability. We get this from a `mutable` or `immutable` modifier on parameters, fields, or return types. A `mutable` modifier always implies a fraction of one. For `immutable`, the precise fraction p is not relevant, it only has to fulfill $0 < p < 1$. We often choose $\frac{1}{2}$ for simplicity.
We uphold the invariant that the sum of the fractions for each id is non-increasing. This means that it is always at most one.
When duplicating a reference by assigning a variable

$$l = m;$$

the fractional permission that l holds is split and one part allocated to m. If the context before the assignment contains $m : \left[\frac{3}{4}\right]_{\text{id\_m}}$, it will contain $l : \left[\frac{3}{8}\right]_{\text{id\_m}}, m : \left[\frac{3}{8}\right]_{\text{id\_m}}$ afterward (note that the proportions of the split do not matter, but we split the permissions by half in this example which is what our implementation does). The splitting is formalized by the Ref-Ref-Assignment type-rule.

3.1.1.1 Recombination

If a variable is not used anymore, we can *move* its permission to another variable of the same id. We call this *recombination* and use *liveness* to determine when we can recombine variables. A variable is *live* at a location in the program, if its current value is used in any other location that is reachable from the current one.
In the previous example, once m is not live anymore, we can move its fraction to l. Assuming the resulting context from above, we then get $l : \left[\frac{3}{8} + \frac{3}{8}\right]_{\text{id\_m}}, m : [0]_{\text{id\_m}}$.
Recombination is formalized in the type system by the RECOMBINATION rule.

3.1.1.2 Mutability and Readability

From the fractional permissions we derive a basic notion of *mutability* and *readability*, which we later refine to *shallow mutability* and *shallow readability*.
Variables are only *mutable* if they are assigned a permission with a fraction of exactly one in the current context. Further, a variable is only *readable* if its fraction is greater than zero.

3.1.2 Tracking Fields: The Borrow List

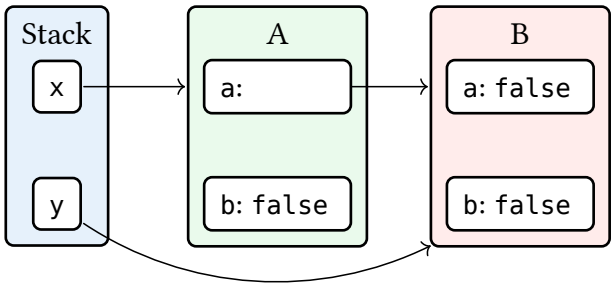
When working with nested objects, we might encounter code like this:

```
class A {
  mutable B a;
  bool b;
}

class B {
  bool a;
  bool b;
}
```

```
A x = ...; // Such that x is mutable
B y = x.a;
```

This results in the following references:



By extracting `x.a` into its own variable `y`, we now have two references to the object of type `B`. Recall that we want to ensure that for mutable access, the reference is unique, so the only readable reference to this object. This creates the following requirements:

1. `y` cannot be mutable, if `x.a` is still readable
2. Similarly, no mutable accesses can be made through `x.a`, if `y` is readable
3. `x.a` and `x.b` should still be reassignable

We fulfill these requirements through the *borrow list*. The borrow list is a collection of *borrows*. Borrows are made up of a source, a fraction, and a target. The source is a combination of an id and a field f. The fraction p is a rational number and the target is an id. We use the following notation for borrows:

$$\text{source\_id}.f \xrightarrow{p} \text{target\_id}$$

A borrow holds the semantic meaning that target_id or a (nested) field of target_id may hold a reference that aliases source_id$.f$ or a (nested) field of it and thus we lose a portion of source_id$.f$'s permission. A permission of 1 on the borrow indicates that the alias can be mutable.

To reason about aliases of source_id itself, we introduce a special *marker field* called *base*. Borrows from source_id$.base$ hold the meaning that there might be an alias of source_id with a different id (the target).

3.1.2.1 Mutability and Readability

Using borrows, we get our final notions of *mutability* and *readability*. We expand our previous concepts, which are only based on the fractional permissions of variables, to take borrows of *base* into account and call them *shallow mutability* and *shallow readability* of a variable v with id id_v:

shallow mutability For v to have shallow mutability, we have to ensure that there are no aliases of v with the same id or with another id. To achieve this, we require that the variable has a permission with fraction 1 (excluding all other variables of the same id from having a non-zero fraction) and that the sum of all fractions of borrows with id_v.*base* as the source is 0.

shallow readability For readability, we only have to ensure that there are no mutable aliases. For this, we require that x has a non-zero fraction and the sum of the fractions

of all borrows with id_v.*base* as the source is smaller than the sum over the fractions of all permissions with id_v as the id[1].

From these *shallow permissions* and the borrows, we define the mutability and readability of a field access $x.f$:

mutable x has *shallow mutability* and the sum of the fractions of all borrows from the field f of its id sum to 0.

readable x has *shallow readability* and the sum of the fractions of all borrows from the field f of its id sum to less than 1 if the field is declared as `mutable` or less than $\frac{1}{2}$ if the field is declared as `immutable`.

From the permissions on field accesses, we define *deep* permissions:

deep mutability All fields of the variable are *mutable*.

deep readability All fields of the variable are *readable*.

Field reassignments are allowed on variables with *shallow mutability*. The values assigned to an `immutable`/`mutable` field need *deep readability/mutability* respectively.

3.1.2.2 Example

In our example above, the Var-From-Field-Assignment, Pseudocall and Pseudoargument type-rules generate a borrow

$$\texttt{x.a} \xrightarrow{p} \texttt{y}$$

and `y` gets a permission $[q]_{\text{id\_y}}$ where $q \in \left\{1, \frac{1}{2}\right\}$ and $q = 1$ implies $p = 1$.

To get a ***mutable*** `y`, we need q to be 1, so p must also be 1. By the definition of *readability* for the field access `x.a`, the borrow causes `x.a` to not be readable while the borrow persists. This case is demonstrated in Listing 3.1.

If we only need a ***readable*** `y`, p only has to be > 0. In this case, `x.a` is still *readable*, but not *mutable* while the borrow is in the borrow list.

```
1 // p : [1]_id_p
2 A x = p;
3 // Splitting: p : [1/2]_id_p, x : [1/2]_id_p
4 // Recombination[2]: p : [0]_id_p, x : [1]_id_p
5 B y = x.a;
6 // p : [0]_id_p  x : [1]_id_p  y : [1]_id_y; borrows: x.a -1-> y
```

Listing 3.1: Borrowing a field mutably

[1]This ensures that there is no borrow with fraction 1. In practice, that is the only restriction that we need but we keep this check more flexible.

[2]Recombination is not technically necessary here as we could also split directly into $[0], [1]$. We demonstrate the case with recombination since this is the way the implementation works.

3.1.2.3 Deleting borrows

In general, we want to delete borrows as soon as it is possible to give back permission to the sources of these borrows.
We reiterate the meaning of a borrow: $x.f \xrightarrow{p} y$ expresses that $x.f$ might be aliased somewhere in y and thus cannot have full permission. Since aliasing is only relevant if the alias is used, we can ignore all aliases that are not live. One might thus consider to delete the above borrow once no variable with id y is live anymore.
But this breaks in the following case: Assume that $x.f$ is aliased by y but $y.g$ is itself aliased by z. This means that $x.f.g$ is also aliased by z, but we don't necessarily have a separate borrow for it We can only delete the $x.f \xrightarrow{p} y$ borrow, and by that give back the permission to $x.f$, if we can be sure that there is no "transitive aliasing". We can conclude this if there is no borrow that has y as the id of its source. In the above case, we would have two borrows:

$$x.f_1 \xrightarrow{p_1} y; \quad y.f_2 \xrightarrow{p_2} z$$

To formalize this, we introduce the concept of *activity* for ids. An id is *active* if any of its variables are live or if there is any borrow with the id in the source.
The use of liveness information instead of the lexical scope of variables for deleting borrows and recombination (Section 3.1.1.1) enables many programs that could not be type-checked otherwise. Rusts borrow checker uses liveness information as well, through a concept called "non-lexical lifetimes" [18].

3.1.2.4 Borrows for function arguments

Our `mutable` and `immutable` annotations are not expressive enough to reason about what a function does to its arguments: As Boyland describes [5, 2.2.1], one shortcoming of a "read-only" qualifier is that a function can still retain a reference to a "read-only" argument, or, more applicable to our case, assign the `immutable` argument to a field on a `mutable` argument or the return value. In that case, after the function call, we would have "duplicated" our reference in the callers context.
As an example, consider the code in Listing 3.2. It shows that after a call, an argument (or a nested field of it) can be aliased by the return value or other arguments and our borrows have to be able to track this. They have to ensure that an argument is not readable while a mutable alias exists and that it is not mutable as long as any alias exists.
We achieve this by creating borrows from an argument's *base* and all reference-typed fields to a special target $\otimes$ which we consider as always active. The fractions of these borrows have to be nonzero, ensuring that arguments never become mutable again, and exactly one, if both the field and parameter are declared as `mutable` (or the field is *base*). In the latter case, there is the possibility that the argument is mutably aliased after the function call, so we ensure that it never becomes readable again.

These rules are quite strong and are currently the main restriction of our type system. In subsection 8.1 we propose two extensions for future work to relax these rules.

```
1  class A {
2    immutable B a;
3    bool b;
4  }
5  class B {
6    bool a;
7    bool b;
8  }
9
10 immutable B callee(mutable A a, immutable B b) {
11   a.a = b;
12   return b;
13 }
14
15 immutable B caller(bool a, bool b) {
16   B b_val = new B(a, b);
17   // b_val : [1]id_b
18   A a_val = new A(new B(false, false), false);
19   // b_val : [1]id_b, a_val : [1]id_a
20   B c = callee(a_val, b_val);
21   // b_val is aliased by a_val.a and c
22   // b_val : [1]id_b, a_val : [1]id_a, c : [1/2]id_c
23   // borrows for a_val: a_val.base -1-> ⊗, a_val.a -1/4-> ⊗
24   // borrows for b_val: b_val.base -1-> ⊗
25   ...
26 }
```

Listing 3.2: Example function, with argument `b` leaving through `a` and the return value. Arguments to `new A(...)` are inlined for better readability.

3.1.3 Enabling Getters: Scopes

As typical Java code uses a lot of getters, our type system should be able to type the following access pattern:

```
1 A a = getA(foo);
2 B b = getB(foo);
```

```
3
4 bar(a,b);
```

In Rust [21], which has a similar type system for references enforcing unique access for mutation, this pattern is not possible. After the `getX` call, foo is not accessible anymore while `x` is live.
We need a way to specify that `getX` only accesses part (the `x` field) of `foo`. Similar ideas have also been discussed for Rust [17], [22].
Our solution is to annotate each parameter with a scope $scope \in 2^{\mathcal{F}}$, $\mathcal{F} = \{\texttt{a}, \texttt{b}, base\}$, $scope \neq \mathcal{F} \setminus \{base\}$. The scope limits which variables are available to a function. We achieve this by inserting borrows $id.f \xrightarrow{p} \otimes$ with p either 1 or $\frac{1}{2}$ for `mutable`/`immutable` parameters into the initial context of a function for each $f \in \mathcal{F} \setminus \{base\}$ that is not in scope. This ensures that these fields are never readable in the function. We never introduce borrows from $base$, since such borrows would limit the *shallow readability/mutability* of the parameter, limiting access to in-scope fields as well.
This could violate type safety if we achieved an alias of a parameter with a different id. However, this is only possible through either a new reference or a function call, respectively with the parameter as an argument, which would require deep readability for the parameter. But our above restriction on scopes ensures that if $base$ is not in scope, then there is at least one other field that is not in scope, preventing the parameter from having deep readability.
Note, however, that out-of-scope fields on `mutable` arguments are still reassignable in the function. While this is compatible with our type safety since all `mutable` arguments are required to have *shallow mutability*, this might be unintuitive, so our implementation does not permit it.
When calling a function with argument x and a scope to a set of fields F we relax our notion of deep mutability/readability from "all fields" to "all fields in F". The function call only introduces borrows for the fields which are in scope.

We can add the following scopes to the signatures of `getX` and `getY` (additional dummy arguments to fulfill our language restriction are omitted):

```
1 mutable A getA(mutable {a} Foo foo)
2 mutable B getB(mutable {b} Foo foo)
```

With these updated signatures, we can correctly type the above example as seen in Listing 3.3.

```
1  class Foo {
2    mutable A a;
3    immutable B b;
4  }
5
6  ...
7
8    Foo foo = new Foo(...);
9    // foo : [1]_id_foo
10   A a = getA(foo);
11   // foo : [1]_id_foo , a : [1]_id_a ; borrows: foo.a -1-> ⊗
12   B b = getB(foo);
13   // foo : [1]_id_foo , a : [1]_id_a , b : [1/2]_id_b ; borrows: foo.a -1-> ⊗, foo.b -1/4-> ⊗
14
15   bar(x,y);
```

Listing 3.3: Example showcasing scopes

3.2 Type Rules

The following type rules formalize the type system given above. They ensure that only unique references can be modified. We provide a proof for this property in section 4.

Our type system uses three contexts:
- T holds the base-types of variables. It does not interact with the permission system.
- Γ holds permissions for local variables and parameters.
- B holds the borrow list.

We define the following functions:

$$\begin{aligned}\mathrm{frac}\big([id]_p\big) &= p\\ \mathrm{id}\big([id]_p\big) &= id\\ \mathrm{id}_\Gamma(x) &= \mathrm{id}(\Gamma(x))\\ \mathrm{frac}_\Gamma(x) &= \mathrm{frac}(\Gamma(x))\end{aligned}$$

To access parts of a variable permission.

$$\begin{aligned}\mathrm{frac}(\texttt{mutable}) &= 1\\ \mathrm{frac}(\texttt{immutable}) &= \frac{1}{2}\end{aligned}$$

To convert mutability modifiers to fractions.

$$\begin{aligned}\text{field-perm}(C\,.\,f) &= \mathrm{frac}(\text{mutability specified for } f \text{ in } C)\\ \text{field-perm}_{T,B}(id_x\,.\,f) &= \text{field-perm}(T(id_x)\,.\,f) - \text{borrow-sum}_B(id_x\,.\,f)\end{aligned}$$

To get the base *fraction* defined by the mutability of a field and the effective fraction for a field access through a specific *id* that remains when subtracting all borrows. We can write $T(id_x)$ because an *id* specifies a set of variables that all alias eachother, so they all have the same base type.

$$\text{borrow-sum}_B(id\,.\,f) = \sum\Big\{q \mid id\,.\,f \xrightarrow{q} \tau \in B\Big\}$$

To get the sum of the *fraction*s of all borrows for a source.

$$\text{readable}_{\text{shallow},B,\Gamma}(l) :=$$
$$\text{frac}_\Gamma(l) \cdot \left(\left(\sum_{\text{id}_\Gamma(l')=\text{id}_\Gamma(l)} \text{frac}_\Gamma(l') \right) - \text{borrow-sum}_B(\text{id}_\Gamma(l).base) \right) > 0$$

$$\begin{aligned}\text{readable}_{\text{deep},T,\Gamma,B}(l) :=\ & \text{readable}_{\text{shallow},B,\Gamma}(l) \\ & \wedge \text{field-perm}_{T,B}(\text{id}_\Gamma(l).\texttt{a}) > 0 \\ & \wedge \text{field-perm}_{T,B}(\text{id}_\Gamma(l).\texttt{b}) > 0\end{aligned}$$

To check readability.

$$\begin{aligned}\text{mutable}_{\text{shallow},B,\Gamma}(l) &:= (\text{frac}_\Gamma(l) \cdot (1 - \text{borrow-sum}_B(\text{id}_\Gamma(l).base)) = 1) \\ \text{mutable}_{\text{deep},T,\Gamma,B}(l) &:= \text{mutable}_{\text{shallow},B,\Gamma}(l) \\ &\quad \wedge \text{field-perm}_{T,B}(\text{id}_\Gamma(l).\texttt{a}) = \text{field-perm}(T(l).\texttt{a}) \\ &\quad \wedge \text{field-perm}_{T,B}(\text{id}_\Gamma(l).\texttt{b}) = \text{field-perm}(T(l).\texttt{b})\end{aligned}$$

To check mutability. We check against the base field-perm instead of 1 to support `immutable` fields as well.

$$\text{base-type}(C.f) = \text{type specified for } f \text{ in } C$$

To get the base type of a field.

For a function fun with definition

$$mut_r\ C\ fun\texttt{(}mut_1\ scope_1\ C_1\ \texttt{a,}\ mut_2\ scope_2\ C_2\ \ \texttt{b) \{}\ \text{stmt}\ \texttt{\}}$$

to be well typed,

$$T_{\text{init}}, \Gamma_{\text{init}}, B_{\text{init}} \vdash \text{stmt} \dashv \bot, \bot, \bot$$

must hold with

$$\begin{aligned}T_{\text{init}} &= \{\texttt{a} : C_1, \texttt{b} : C_2\} \\ \Gamma_{\text{init}} &= \left\{\texttt{a} : [\text{frac}(mut_1)]_\text{a}, \texttt{b} : [\text{frac}(mut_2)]_\text{b}\right\} \\ B_{\text{init}} &= \text{init-borrows}(\texttt{a}, C_1, scope_1) \cup \text{init-borrows}(\texttt{b}, C_2, scope_2) \\ \text{init-borrows}(arg, C, scope) &= \left\{\text{arg}\ .f \xrightarrow{\text{field-perm}_{T,B}(C.f)} \otimes \;\middle|\; f \in scope\right\}\end{aligned}$$

3.2.1 Declarations

In our formalization language, variables have to be declared with their type before they are used.

3.2.1.1 Ref-Decl

$$\frac{T' = T \lhd \{l : C\}}{T, \Gamma, B \vdash C\ l;\ \dashv T', \Gamma, B}\ \textsc{Ref-Decl}$$

Declaring a new variable of a reference type inserts the declared type into the base type context.

3.2.1.2 Bool-Decl

$$\frac{T' = T \lhd \{b : \texttt{bool}\}}{T, \Gamma, B \vdash \texttt{bool}\ b;\ \dashv T', \Gamma, B}\ \textsc{Bool-Decl}$$

Declaring a new boolean variable inserts the `bool` type into the base type context.

3.2.2 Assignments

The following rules type simple assignments between two reference-typed variables (where our fractional permissions are applied) and assignments of a boolean literal to a variable.

3.2.2.1 Ref-Ref-Assignment

$$\frac{T(x) = T(y) \qquad \Gamma(y) = [p]_{\text{id}} \qquad \Gamma' = \Gamma \lhd \left\{x : [q]_{\text{id}},\ y : [p']_{\text{id}}\right\} \qquad p = p' + q}{T, \Gamma, B \vdash x = y;\ \dashv T, \Gamma', B}\ \textsc{Ref-Ref-Assignment}$$

A basic assignment between two reference-typed variables. The fraction of the source is split to the target.

3.2.2.2 Bool-Lit-Assignment

$$\frac{T(b) = \texttt{bool}}{T, \Gamma, B \vdash b = \text{bool-literal};\ \dashv T, \Gamma, B}\ \textsc{Bool-Lit-Assignment}$$

Assignment of a boolean literal to a boolean variable.

3.2.3 Call-Like

In our language, we have several statements that need similar premises during typechecking and produce similar results. We abstract these to a concept we call a "pseudocall". For Var-From-Field-Assignment and Field-Assignment this can be seen as transforming these statements to the corresponding getter or setter.
A Pseudocall contains the mutability of its return value, the base type it returns and two pseudo arguments.
Pseudoargs contain the mutability of the parameter, a borrow type, the scope of the paramter, the base type of the paramter and the argument itself. The borrow type is either "returnval" or "persistent". Arguments with a persistent borrow type create borrows to $\otimes$, arguments with returnval create borrows with the new id of the variable holding the returned value as the target. This enables us to create borrows that we can delete immediately after the pseudocall is checked, in cases where we know this to be sound. We use such borrows in the Var-From-Field-Assignment and Ref-Creation rules.

3.2.3.1 Function-Call

$$\frac{\begin{array}{l}\text{sig}(fun) = mut_1\ C_1\ fun(mut_2\ scope_2\ C_2, mut_3\ scope_3\ C_3) \\ \quad T, \Gamma, B \vdash l = \text{Pseudocall}(\\ \quad\quad mut_1, \\ \quad\quad C_1, \\ \quad\quad \text{Pseudoarg}(mut_2, \text{persistent}, scope_2, C_2, m), \\ \quad\quad \text{Pseudoarg}(mut_3, \text{persistent}, scope_3, C_3, n) \\ \quad); \dashv T, \Gamma, B'\end{array}}{T, \Gamma, B \vdash l = fun(m, n); \dashv T, \Gamma, B'}\ \textsc{Function-Call}$$

For a function call, the translation to the pseudocall is trivial.

3.2.3.2 Ref-Creation

$$\frac{\begin{array}{l}T, \Gamma, B \vdash l = \text{Pseudocall}(\\ \quad \texttt{mutable}, \\ \quad C, \\ \quad \text{Pseudoarg}(\text{mutability}(C.\texttt{a}), \text{returnval}, \{base, \texttt{a}, \texttt{b}\}, \text{type}(C.\texttt{a}), m), \\ \quad \text{Pseudoarg}(\text{mutability}(C.\texttt{b}), \text{returnval}, \{base, \texttt{a}, \texttt{b}\}, \text{type}(C.\texttt{b}), n) \\ ); \dashv T, \Gamma, B'\end{array}}{T, \Gamma, B \vdash l = \texttt{new } C(m,n); \dashv T, \Gamma, B'} \text{ Ref-Creation}$$

For the creation of a new reference, we transform the statement to a pseudocall with a `mutable` return value, the mutabilities of the fields as the argument mutabilities and full scopes.

3.2.3.3 Field-Assignment

$$\frac{\begin{array}{l}T, \Gamma, B \vdash \_ = \text{Pseudocall}(\\ \quad \texttt{mutable}, \\ \quad \texttt{bool}, \\ \quad \text{Pseudoarg}(\texttt{mutable}, \text{returnval}, \{base\}, T(l), l), \\ \quad \text{Pseudoarg}(\text{mutability}(C.f), \text{persistent}, \{base, \texttt{a}, \texttt{b}\}, \text{type}(C.f), m) \\ ); \dashv T, \Gamma, B'\end{array}}{T, \Gamma, B \vdash l.f = m; \dashv T, \Gamma, B'} \text{ Field-Assignment}$$

To type check an assignment to a field, we transform it to a pseudocall that represents a setter for this field in its type signature.

3.2.3.4 Var-From-Field-Assignment

$$\frac{\begin{array}{l} mut \in \{\texttt{immutable}, \texttt{mutable}\} \\ T, \Gamma, B \vdash l = \text{Pseudocall}(\\ \quad mut, \\ \quad \text{base-type}(T(m).f), \\ \quad \text{Pseudoarg}(mut, \text{returnval}, \{f\}, T(m), m), \\ \quad \text{Pseudoarg}(\texttt{immutable}, \text{persistent}, \emptyset, \texttt{bool}, \texttt{false}) \\ ); \dashv T, \Gamma, B' \end{array}}{T, \Gamma, B \vdash l = m.f; \dashv T, \Gamma, B'} \text{Var-From-Field-Assignment}$$

To type check an assignment to a field, we transform it to a pseudocall that represents a getter for this field in its type signature.

3.2.4 Pseudocalls

These rules are used to type-check the psedocalls generated by all call-like statements.

3.2.4.1 Pseudocall

$$\frac{\begin{array}{c} T, \Gamma, B \vdash id_{\text{ret}} \leftarrow \text{Pseudoarg}(mut_{\text{arg1}}, borrowtype_1, scope_{\text{arg1}}, \tau_1, a) \dashv T, \Gamma', B' \\ T, \Gamma', B' \vdash id_{\text{ret}} \leftarrow \text{Pseudoarg}(mut_{\text{arg2}}, borrowtype_2, scope_{\text{arg2}}, \tau_2, b) \dashv T, \Gamma'', B'' \\ \Gamma''' = \Gamma'' \lessdot \left\{ l : [\text{frac}(mut_{\text{ret}})]_{id_{\text{ret}}} \;\middle|\; \tau_{\text{ret}} \neq \texttt{bool} \right\} \qquad id_{\text{ret}} \text{ is a fresh id} \end{array}}{\begin{array}{l} T, \Gamma, B \vdash l = \text{Pseudocall}(\\ \quad mut_{\text{ret}}, \\ \quad \tau_{\text{ret}}, \\ \quad \text{Pseudoarg}(mut_{\text{arg1}}, borrowtype_1, scope_{\text{arg1}}, \tau_1, a), \\ \quad \text{Pseudoarg}(mut_{\text{arg2}}, borrowtype_2, scope_{\text{arg2}}, \tau_2, b) \\ ); \dashv T, \Gamma''', B'' \end{array}} \text{Pseudocall}$$

For typing the pseudocall, we type the arguments in order with chained contexts and then insert a new permission for the variable holding the return value with a new id and a fraction of either 1 or $\frac{1}{2}$, depending on whether the return value is specified as `mutable` or `immutable`.

3.2.4.2 Pseudoargument

$$\frac{T(x) = \texttt{bool}}{T, \Gamma, B \vdash \text{Pseudoarg}(\texttt{immutable}, type_{borrow}, \emptyset, \texttt{bool}, x) \dashv T, \Gamma, B} \textsc{Pseudoarg-Bool}$$

$$\frac{\begin{array}{c}\text{field-perm}_{T,B}(\text{id}_\Gamma(x).f) = 1 \\ B' = B \cup \left\{\text{id}_\Gamma(x).f \xrightarrow{1} borrowtarget\right\}\end{array}}{T, \Gamma, B \vdash (\texttt{mutable}, x.f, borrowtarget) \dashv T, \Gamma, B'} \textsc{Pseudoarg-Field-Mut}$$

$$\frac{\begin{array}{c}p_f = \text{field-perm}_{T,B}(\text{id}_\Gamma(x).f) \qquad p_f > 0 \\ B' = B \cup \left\{\text{id}_\Gamma(x).f \xrightarrow{\frac{p_f}{2}} borrowtarget\right\}\end{array}}{T, \Gamma, B \vdash (\texttt{immutable}, x.f, borrowtarget) \dashv T, \Gamma, B'} \textsc{Pseudoarg-Field-Immut}$$

$$\frac{\begin{array}{c}T(x) = \tau \qquad \text{readable}_{\text{shallow},B,\Gamma}(x) \\ mut_{\text{param}} = \texttt{mutable} \implies \text{mutable}_{\text{shallow},B,\Gamma}(x) \qquad scope \neq \{\texttt{a}, \texttt{b}\} \\ \forall f \in scope : T, \Gamma, B \vdash \left(mut_{\text{param}}, x.f, \text{target}(type_{borrow})\right) \dashv T, \Gamma, B_f \\ B' = \bigcup_{f \in scope} B_f\end{array}}{T, \Gamma, B \vdash id_{\text{ret}} \leftarrow \text{Pseudoarg}\left(mut_{\text{param}}, type_{borrow}, scope, \tau, x\right) \dashv T, \Gamma, B'} \textsc{Pseudoargument}$$

With

$$\text{target}(\text{persistent}) = \otimes$$
$$\text{target}(\text{returnval}) = id_{\text{ret}}$$

For boolean (or generally primitive, `bool` is our only primitive type) arguments, there does not need to be any type checking, apart from the base type, as they are always immutable. When type checking a (pseudo) argument, we first validate that its shallow mutability or readability fits the declared mutability of the parameter. We then quantify over the scope and validate that all in-scope fields have the necessary deep permissions, and insert the borrows.

3.2.5 Return-Statement

$$\frac{\begin{array}{c}\text{sig}(\text{current function}) = mut_r\ \tau_r\ fun(..., ...) \qquad T(l) = \tau_r \\ \text{readable}_{\text{deep},T,\Gamma,B}(l) \qquad mut_r = \texttt{mutable} \rightarrow \text{mutable}_{\text{deep},T,\Gamma,B}(l)\end{array}}{T, \Gamma, B \vdash \texttt{return}\ l; \dashv \bot, \bot, \bot} \textsc{Return-Statement}$$

$$\frac{\text{sig}(\text{current function}) = \texttt{bool}\ \ fun(\ldots,\ldots) \qquad T(l) = \texttt{bool}}{T, \Gamma, B \vdash \texttt{return}\ l\texttt{;} \dashv \bot, \bot, \bot}\ \textsc{Bool-Return-Statement}$$

When returning booleans, we do not have any additional requirements.
When returning a value of a reference type, we have to ensure that it is deep readable, and, if the return type of the function is specified as `mutable`, that it is deep mutable as well.

3.2.6 Control Flow

Our language has only two types of control flow:
- An if-else-branch on a `bool` variable
- Sequential control flow

3.2.6.1 If-Statement

$$\frac{\begin{array}{c} T, \Gamma, B \vdash s_1 \dashv T, \Gamma_\alpha, B_\alpha \qquad T, \Gamma, B \vdash s_1 \dashv T, \Gamma_\beta, B_\beta \\ \Gamma' = \text{merge-perms}(\Gamma_\alpha, \Gamma_\beta) \qquad B' = \text{merge-borrows}(B_\alpha, B_\beta) \end{array}}{T, \Gamma, B \vdash \texttt{if (}b\texttt{) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}} \dashv T, \Gamma', B'}\ \textsc{If-Statement}$$

We define functions to merge the permission context:

$$\text{merge-perms}(\mathrm{A}, \mathrm{B}) := \{l : \text{take-lower}(\mathrm{A}(l), \mathrm{B}(l)) \mid l \in (\text{domain}(A) \cap \text{domain}(B))\}$$

$$\text{take-lower}\left([\alpha]_{\text{id}_\alpha}, [\alpha]_{\text{id}_\alpha}\right) := \begin{cases} \min\{\alpha, \beta\} & \mid \text{id}_\alpha = \text{id}_\beta \\ \top & \mid \text{else} \end{cases}$$

and functions to merge the borrow context:

$$\text{merge-borrows}(\mathrm{A}, \mathrm{B}) := \text{merge}(\mathrm{A} \cup \mathrm{B})$$

$$\text{merge}(S) := \left\{\text{choose-highest}(\sigma, \tau, S) \;\middle|\; \sigma \xrightarrow{p} \tau \in S\right\}$$

$$\text{choose-highest}(\sigma, \tau, S) := \sigma \xrightarrow{\max\left\{p \;\middle|\; \sigma \xrightarrow{p} \tau \in S\right\}} \tau$$

For an if-statement, we type-check both branches separately and then combine Γ and B as follows: Combine permissions for a variable by taking the minimum *fraction* if both permissions have the same *id*, if they have different *id*s, assign $\top$.
Borrows with the same source and target are combined by choosing the borrow with the higher value.

3.2.6.2 Sequential

$$\frac{T, \Gamma, B \vdash s_1 \dashv T', \Gamma', B' \qquad T', \Gamma', B' \vdash s_2 \dashv T'', \Gamma'', B''}{T, \Gamma, B \vdash s_1 \; s_2 \dashv T'', \Gamma'', B''} \textsc{Sequential}$$

For chained execution, we just chain the type-checking.

3.2.7 Redistribution

These following rules all "give back" permission through deleting borrows or recombination.
For borrow deletion we introduce the DeleteBorrows(set-of-borrows-to-delete) helper.

3.2.7.1 Null-Borrow-Deletion

$$\frac{}{T, \Gamma, B \vdash \text{DeleteBorrows}(\emptyset) \dashv T, \Gamma, B} \textsc{Null-Borrow-Deletion}$$

Base case for borrow deletion.

3.2.7.2 Borrow-Deletion

$$\frac{b = \text{var\_id}.f \xrightarrow{p} \text{borr\_id} \qquad B' = B \setminus \{b\} \qquad T, \Gamma, B' \vdash \text{DeleteBorrows}(B_{\text{rem}}) \dashv T'', \Gamma'', B''}{T, \Gamma, B \vdash \text{DeleteBorrows}(\{b\} \cup B_{\text{rem}}) \dashv T'', \Gamma'', B''} \textsc{Borrow-Deletion}$$

To delete a borrow with a field in its source, we can just remove it from the borrow list.

3.2.7.3 Inactive-Borrow-Deletion

$$\frac{b \in B \qquad b = \sigma \xrightarrow{p} \text{borr\_id} \qquad \text{borr\_id not active} \qquad T, \Gamma, B \vdash \text{DeleteBorrows}(\{b\}) \dashv T', \Gamma', B' \qquad T', \Gamma', B' \vdash s \dashv T'', \Gamma'', B''}{T, \Gamma, B \vdash s \dashv T'', \Gamma'', B''} \textsc{Inactive-Borrow-Deletion}$$

We define activity as follows:

$$id \text{ is active} \iff \nexists \text{ local variable } v : \mathrm{id}_\Gamma(v) = v \land v \text{ is live}$$
$$\land \forall\ x.f \xrightarrow{p} \tau : x \neq id$$

3.2.7.4 Recombination

$$\frac{\begin{array}{c} l \text{ not live} \qquad \mathrm{id}_\Gamma(l) = \mathrm{id}_\Gamma(m) \qquad p_l = \mathrm{frac}_\Gamma(l) \qquad p_m = \mathrm{frac}_\Gamma(m) \\ \Gamma' = \Gamma \Lhd \left\{ m : [p_l + p_m]_{\mathrm{id}_\Gamma(m)},\ l : [0]_{\mathrm{id}_\Gamma(l)} \right\} \\ T, \Gamma', B \vdash s \dashv T'', \Gamma'', B'' \end{array}}{T, \Gamma, B \vdash s \dashv T'', \Gamma'', B''}\ \textsc{Recombination}$$

When a variable is not live live anymore, we can recombine its permission with another variable of the same id.

3.2.8 Program entry

At the entry point of a program, each object on the heap must only be referenced by exactly one other heap object or parameter.

4 Type Safety

In this section, we prove that our type system actually ensures the property we want, namely that mutation is only possible with exclusive access. We formalize this as a property we call "conforming" (Definition 4.2) and prove that it holds for every reachable state in Theorem 4.18 (Reachable Conforming).

Definition 4.1 **Assignable** We call $x.f$ *assignable* if and only if x has *shallow mutability*.

Definition 4.2 **Conforming State** σ, ℓ is conforming to the context (T, Γ, B) (we write $\sigma, \ell : (T, \Gamma, B)$)

$\iff$

If v is a live local variable or parameter that references an object o on the heap and $o.f$ is readable through v, then there is no live variable or parameter $w \neq v$ that references o where $w.f$ is assignable.

Definition 4.3 **Live Variables** A variable is live $\iff$ its current value will be read in any following state.

Definition 4.4 **Live Id** An id is live $\iff$ there is a *live variable* with that id.

Definition 4.5 **Active Id** An id x is active $\iff$ it is *live* or there is a borrow $x.f \xrightarrow{p} \tau$.

Definition 4.6 **Id Path Equivalence** We write $x.f_1.\cdots.f_n \equiv y.g_1.\cdots.g_m$ $(n, m \in \mathbb{N})$ $\iff$ $x.f_1.\cdots.f_n$ and $y.g_1.\cdots.g_m$ reference the same object in the state σ, ℓ. Formally:

$$
\begin{aligned}
x \equiv y &\iff \sigma(heap)(x) = \sigma(heap)(y) \\
x.f \equiv y &\iff \sigma(heap)(x)(f) = \sigma(heap)(y) \\
x.f_1.\cdots.f_n \equiv y &\iff x.f_1.\cdots.f_{n-1} \equiv z \wedge z.f_n \equiv y \\
x \equiv y.g_1.\cdots.g_m &\iff y.g_1.\cdots.g_m \equiv x
\end{aligned}
$$

We write $(x.f_1.\cdots.f_n$ in initial state$)$ where x is a parameter id to refer to the object x refers to in the functions initial state. Formally, we use σ_{init} instead of σ when evaluating the heap.

Definition 4.7 **Permission Sum** Permission sum of id:

$$\text{perm-sum}(id) := \sum \{p \mid l : [p]_{id} \in \Gamma\}$$

Definition 4.8 **Field Access Readability** We call $x.f$ "readable" iff x has shallow readability and field-perm$_{T,B}(x.f) > 0$

Definition 4.9 **Field Access Mutability** We call $x.f$ "mutable" iff x has shallow mutability and field-perm$_{T,B}(x.f) = 1$

To prove the type safety of our system, we first define some invariants that hold in every reachable state.

Definition 4.10 **Monotone Permission Sum** The *permission sum* for each id is ≤ 1 and monotonically decreasing throughout any execution through a program.

Definition 4.11 **Mut Exclusivity** Let x, y be *active* ids with $x.f_1.\cdots.f_n \equiv y.g_1.\cdots.g_m$ and $x.f_1.\cdots.f_n \neq y.g_1.\cdots.g_m$.

1. If $x.f_1$ is mutable and $f_1, ..., f_n$ are declared as `mutable`, then as long as x is *active*, $y.g_1$ is not readable.
2. If $x.f_1$ is readable, then as long as x is *active*, $y.g_1$ is not mutable or one of $g_1, ..., g_m$ is declared as `immutable`.

If $n = 0$, then the mutability/readability requirements on $x.f_1$ are replaced by $x.base$ and we say that $x.base$ is mutable/readable iff x has shallow mutability/readability. The same applies to y.

We now prove that our invariants are preserved across statements that are correctly typed. We use σ, T, Γ, B for the pre-state and $\tilde{\sigma}$, $\tilde{T}$, $\tilde{\Gamma}$, $\tilde{B}$ for the post-state.

Lemma 4.12 **Monotone Permission Sum Preservation** Let σ and T, Γ, B be a reachable state and context combination. If *Monotone Permission Sum* holds for σ and T, Γ, B, and

$$T, \Gamma, B \vdash s \dashv \tilde{T}, \tilde{\Gamma}, \tilde{B}$$
$$\sigma \overset{s}{\rightsquigarrow} \tilde{\sigma}$$

then *Monotone Permission Sum* holds for $\tilde{\sigma}$ and $\tilde{T}, \tilde{\Gamma}, \tilde{B}$.

Proof. To show the ≤ 1 bound, we consider new ids. New ids are only introduced in the initial context of a fuction or by the PSEUDOCALL rule. In both cases, the fraction for the id is the value of frac(mut) with $mut \in \{$`mutable`, `immutable`$\}$ which is either 1 or $\frac{1}{2}$, so ≤ 1.

We now have to show that the *permission sum* of an id stays constant over each rule application:

When assigning a variable v a permission with a different id, the *permission sum* decreases:

$$\text{perm-sum}_{\tilde{\Gamma}}(id) = \text{perm-sum}_{\Gamma}(id) - \text{id}_{\Gamma}(v) \leq \text{perm-sum}_{\Gamma}(id)$$

There are three rules that change the permission fraction on an existing variable:

- Ref-Ref-Assignment:

$$\begin{aligned} & \text{perm-sum}_{\tilde{\Gamma}}(id) \\ = \ & \text{perm-sum}_{\Gamma}(id) - p + (q + p') \\ = \ & \text{perm-sum}_{\Gamma}(id) - (q + p') + (q + p') \\ = \ & \text{perm-sum}_{\Gamma}(id) \end{aligned}$$

- Recombination:

$$\begin{aligned} & \text{perm-sum}_{\tilde{\Gamma}}(\text{id}_{\Gamma}(m)) \\ = \ & \text{perm-sum}_{\Gamma}(\text{id}_{\Gamma}(m)) \\ & - \text{frac}_{\Gamma}(m) - \text{frac}_{\Gamma}(l) + \text{frac}_{\tilde{\Gamma}}(m) + \text{frac}_{\tilde{\Gamma}}(l) \\ = \ & \text{perm-sum}_{\Gamma}(\text{id}_{\Gamma}(m)) \\ & - \text{frac}_{\Gamma}(m) - \text{frac}_{\Gamma}(l) + (\text{frac}_{\Gamma}(m) + \text{frac}_{\Gamma}(l)) + 0 \\ = \ & \text{perm-sum}_{\Gamma}(\text{id}_{\Gamma}(m)) \end{aligned}$$

- If-Statement:

  For every id x, we assume by induction:

$$\begin{aligned} \text{perm-sum}_{\Gamma_\alpha}(x) &\leq \text{perm-sum}_{\Gamma}(x) \\ \text{perm-sum}_{\Gamma_\beta}(x) &\leq \text{perm-sum}_{\Gamma}(x) \end{aligned}$$

  We have

$$\begin{aligned} & \text{perm-sum}_{\tilde{\Gamma}}(x) \\ = \ & \sum \left\{ \min\{p_\alpha, p_\beta\} \;\middle|\; l : [p_\alpha]_{\text{id}_\Gamma} \in \Gamma_\alpha \wedge l : [p_\beta]_{\text{id}_\Gamma} \in \Gamma_\beta \right\} \\ \leq \ & \sum \left\{ p_\alpha \mid l : [p_\alpha]_{\text{id}_\Gamma} \in \Gamma_\alpha \right\} \\ = \ & \text{perm-sum}_{\Gamma_\alpha}(x) \end{aligned}$$

  and by the identical argument $\text{perm-sum}_{\tilde{\Gamma}}(x) \leq \text{perm-sum}_{\Gamma_\beta}(x)$.

□

Corollary 4.13 **Monotone Permission Sum All Reachable** *Monotone Permission Sum* holds in all reachable states.

Proof. In the program entry, *Monotone Permission Sum* holds since parameters get a permission with fraction 1 or $\frac{1}{2}$. Using Lemma 4.12 (Monotone Permission Sum Preservation), we get that it must hold in all reachable states. □

Lemma 4.14 **Borrow Preservation** Let (T, Γ, B), $(\tilde{T}, \tilde{\Gamma}, \tilde{B})$ be valid contexts with $(T, \Gamma, B) \vdash s \dashv (\tilde{T}, \tilde{\Gamma}, \tilde{B})$.
Let x be an *active* id, f a field on the type of x, y an *active* id or $\otimes$ (in which case we count it as always active) and let there be a borrow $x.f \xrightarrow{p} y \in B$.

Then as long as y is *active*, there is a borrow $\sigma.f \xrightarrow{q} y$ with $q \geq p$.

Proof. *Activity* is a monotonic property, as once an id is not *active* anymore, there are no *live* variables with this id anymore and thus there can be no new variable with this id and no new borrow with this id occuring in the source.

We consider the type rules that can modify the borrow:

- Inactive-Borrow-Deletion: This rule deletes borrows only when y is not *active* anymore so it does not violate our conclusion.
- If-Statement: If y is *active* after the if-statement, it is *active* for the whole true-branch or the whole else-branch. By induction we get that there is a borrow $x.f \xrightarrow{p_\alpha} y$ with $p_\alpha \geq p$ in the resulting borrow list of this branch. Since the context merging of the if takes the maximum borrow for each (source, target) pair, there will be a $x.f \xrightarrow{q} y$ with $q \geq p_\alpha \geq p$ in the final borrow list. □

This next lemma states that if we hold a mutable reference in a function, that is available outside of the function (meaning it was passed in an argument), that it was actually passed through a `mutable` argument (and only through `mutable` fields on that argument). This ensures that the borrow created by Pseudoargument holds fraction 1 and thus makes the object references by the mutable reference inaccessible through any pre-existing refernce in the caller.

Lemma 4.15 **Mutable Parameter Source** Let x be an id with shallow mutability, p a parameter with id α, $x.f_1.\cdots.f_n \equiv (\alpha.g_1.\cdots.g_m$ in initial state), $f_1, ..., f_n$ declared as `mutable`, $x.f_1$ readable and $\alpha.f_1$ in scope.

Then there must exist a mutable parameter q with id β, $x.f_1.\cdots.f_n \equiv (\beta.h_1.\cdots.h_i$ in initial state), $h_1, ..., h_i$ declared as `mutable` and h_1 in scope.

Proof. We do a case distinction over the type rules for statements that can establish the equivalence relationship:

- It can be established by a side effect or return value of a function call (Function-Call). In this case, a reference to $\alpha.g_1.\cdots.g_m$'s initial state must be passed to the function and by induction over the callees execution, it must be passed mutably, so the argument must have a mutable id y with $y.k_1.\cdots.k_j \equiv (\alpha.g_1.\cdots.g_m$ in initial state), $k_1, ..., k_j$ declared as `mutable` and $y.k_1$ readable.
- The return value of Ref-Creation with return id x or Field-Assignment with receiver id x: Since f_1 is declared as `mutable`, there must be a `mutable` argument with id y in the generated pseudocall. Since in the postcontext, $x.f_1 \equiv y$, in the precontext, $y.f_2.\cdots.f_n \equiv (\alpha.g_1.\cdots.g_m$ in initial state) and $y.f_2$ is readable.
- Var-From-Field-Assignment with target variable id x. Then the receiver id y must have shallow mutability, field f must be declared as `mutable` and $x.f_1.\cdots.f_n \equiv y.f.f_1.\cdots.f_n \equiv (\alpha.g_1.\cdots.g_m$ in initial state)

In all three cases, the id y fulfills the requirements for us to apply induction over the prior execution. □

Lemma 4.16 **Mut Exclusivity Preservation** Let σ and T, Γ, B be a reachable state and context combination. If *Mut Exclusivity* holds for σ and T, Γ, B, and

$$T, \Gamma, B \vdash s \dashv \tilde{T}, \tilde{\Gamma}, \tilde{B}$$
$$\sigma \overset{s}{\rightsquigarrow} \tilde{\sigma}$$

then *Mut Exclusivity* holds for $\tilde{\sigma}$ and $\tilde{T}, \tilde{\Gamma}, \tilde{B}$.

Proof. As this proof uses Lemma 4.14 (Borrow Preservation) in every case, we do not mention every application of it explicitly.

The statements that can establish a new equivalence $x.f_1.\cdots.f_n \equiv y.g_1.\cdots.g_m$ in the state are exactly the ones typed by the call-like type rules in Section 3.2.3:

Function-Call

We first show, that Mut Exclusivity (Definition 4.11) holds in the initial state of the callee, so that we can apply induction over its execution. We do this by contradiction:

Let α, β, be ids of parameters with respective argument ids x, y and $\alpha.f_1.\cdots.f_n \equiv \beta.g_1.\cdots.g_m$.

Assume $\alpha.f_1$ is mutable, $f_1, ..., f_n$ are all declared `mutable`, and $\beta.g_1$ is readable in one following context. Then $\beta.g_1$ must also be readable in the initial context. Since α, β are parameter ids, $x.f_1.\cdots.f_n \equiv y.g_1.\cdots.g_m$ must hold in the caller's context, $x.f_1$ must be mutable, and $y.g_1$ must be readable. This is a contradiction to our assumptions.

Assume $\alpha.f_1$ is readable, $\beta.g_1$ is mutable in one following context, and $g_1, ..., g_m$ are all declared `mutable`. Then $\beta.g_1$ must also be readable in the initial context. Since α, β are parameter ids, $x.f_1.\cdots.f_n \equiv y.g_1.\cdots.g_m$ must hold in the caller's context,

$x.f_1$ must be readable, and $y.g_1$ must be mutable. This is a contradiction to our assumptions.

Next, we show that if a reference to $x.f_1.\cdots.f_n$ is changed by a side effect in a function call, no prior reference to it is readable afterwards.
To modify it in any way, there must be a reference to $x.f_1.\cdots.f_n$ passed in an argument.

- If it is not changed by a nested function call, we can assume, without loss of generality, that f_n has the field that is reassigned in the function.
  By the Field-Assignment and Pseudoargument rules, in the state where the field is reassigned, we have an id α with shallow mutability. Since $\alpha \equiv x.f_1.\cdots.f_n$ and and there is a reference to $x.f_1.\cdots.f_n$ passed in an argument, we can apply Lemma 4.15 (Mutable Parameter Source) and get that there must be a mutable parameter with g_1 in scope whose argument with id y references $x.f_1.\cdots.f_n$ mutably: $x.f_1.\cdots.f_n \equiv y.g_1.\cdots.g_m$, $y.g_1$ is mutable, $g_1, ..., g_m$ are all declared as `mutable`.
- By applying the previous argument inductively over nested function calls, we get an identical y for functions that modify $x.f_1.\cdots.f_n$ only in a nested call.

By induction, we get that before the function call, for each reference $z.h_1.\cdots.h_i \equiv x.f_1.\cdots.f_n \equiv y.g_1.\cdots.g_m$ with $z.h_1.\cdots.h_i \neq y.g_1.\cdots.g_m$, $z.h_1$ is not readable as long as y is *active*.
From the Pseudoargument and Function-Call rules we get that there will be a borrow

$$y.g_1 \xrightarrow{1} \otimes$$

in the post-context. Since this keeps y *active* and $y.g_1$ itself not readable indefinitely, we get that no preexisting reference to $x.f_1.\cdots.f_n$ is readable afterwards.

If one of $x.f_1$, $y.g_1$ is not readable, both conditions of Mut Exclusivity (4.11) trivially fulfilled. By the above proof, we thus assume from now on that if x, y are not new ids, that $x.f_1.\cdots.f_n$ and $y.g_1.\cdots.g_m$ are not modified.

A function call l `= fun(m,n);` can establish the equivalence in the following ways:

- v_x `= fun(a,b);` with $x = y$:
  Then, for the id of the return value r, in the return state of the function, $r.f_1.\cdots.f_n \equiv r.g_1.\cdots.g_m$ must hold.

  If $x.f_1$ is mutable and $f_1, ..., f_n$ are declared as `mutable`, $r.f_1$ must be mutable and by induction, $r.g_1$ is not readable. This case is not allowed by the Return-Statemen-trule.

  If $x.f_1$ is readable then $r.f_1$ must be readable and by induction, $r.g_1$ is not mutable or not all of $g_1, ..., g_m$ are declared as `mutable`.
  If $g_1, ..., g_m$ are all declared as mutable and $r.g_1$ is not mutable, then for $r.g_1$ to not be mutable r cannot have deep mutability and by the Return-Statement rule the

return value is declared as `immutable`. Then $x = y$ has *permission sum* $\frac{1}{2}$ and this $y.g_1$ is never mutable.

- v_x `=` fun`(`v_z`,`b`);` with $x \neq y$ so y not a new id:
  Then for the id of the return value r, in the return state of the function, $r.f_1.\cdots.f_n \equiv y.g_1.\cdots.g_m$. Since we assume that $y.g_1.\cdots.g_m$ is not changed by the function, a reference equivalent to $y.g_1.\cdots.g_m$ must be passed to the function in an argument where g_1 is in scope.

  If the return value is declared as mutable and all of $f_1, ..., f_n$ is declared as `mutable` there must be a `mutable` parameter that references $y.g_1.\cdots.g_m$ mutably.

  Let the argument for this parameter have id z. We then have $z.h_1.\cdots.h_i \equiv y.g_1.\cdots.g_m$, and a borrow in the post-context:

  $$z.h_1 \xrightarrow{p} \otimes$$

  with $p > 0$ and if $f_1, ..., f_n$ are all declared mutable and x has shallow mutability then $p = 1$.

  If $x.f_1$ is mutable and $f_1, ..., f_n$ are all mutable, then $p = 1$ so $z.h_1$ is never readable in any follwing context. If $y.g_1.\cdots.g_m \neq z.h_1.\cdots.h_i$, then since Mut Exclusivity (Definition 4.11) holds in the pre-context/state, $z.h_1$ is mutable in the pre-context and $h_1, ..., h_i$ are all declared as `mutable`, $y.g_1$ is not readable as long as z is *active*. Because the borrow can never be deleted, z is *active* in all following context, so $y.g_1$ is never readable anymore.

  If $x.f_1$ is readable, we can only assume $p > 0$. This ensures $z.h_1$ is not mutable in any following context. Because $z.h_1$ is readable in the pre-context/state and Mut Exclusivity (Definition 4.11) holds, $y.g_1$ is not readable as long as z is *active* or $g_1, ..., g_m$ are not all declared as `mutable`. Since z is *active* in all following context, $y.g_1$ is never readable anymore or not all of $g_1, ..., g_m$ are declared as `mutable`.

- v_y `=` fun`(`v_z`,`b`);` with $x \neq y$ so y not a new id:
  This is the previous case just with x and y swapped.

  If $x.f_1$ is mutable with $f_1, ..., f_n$ declared as `mutable`, then by the previous case we get that $y.g_1$ cannot be readable. This is not possible with the Pseudocall rule.

  If $x.f_1$ is readable, then by the by the previous case, not all of $g_1, ..., g_m$ are declared as `mutable` or $y.g_1$ is not mutable. In the second case, the Pseudocall rule ensures this only possible, if y has a *permission sum* < 1. Using Corollary 4.13 (Monotone Permission Sum All Reachable) we get that y can never get shallow mutability, so $y.g_1$ can never become mutable.

The remaining case, where neither x nor y are the return is not relevant since we assume that neither $x.f_1.\cdots.f_n$ or $y.g_1.\cdots.g_m$ are changed by the function and thus the equivalence also exists in the pre-context/state where Mut Exclusivity (Definition 4.11) already holds.

Ref-Creation

We again have a few cases of how the equivalence can be established:

- v_x `= new` $C(a,b)$`;` with $x = y$:

  Then there must be parameters with ids z for f_1, w for g_1 with $x.f_1.\cdots.f_n \equiv z.f_2.\cdots.f_n$ and $x.g_1.\cdots.g_m \equiv w.g_2.\cdots.g_m$, so also $z.f_2.\cdots.f_n \equiv w.g_2.\cdots.g_m$.

  By the Ref-Creation and Pseudoargument rules, there will be borrows

  $$z.base \xrightarrow{p} x$$
  $$w.base \xrightarrow{q} x$$

  If $x.f_1$ is mutable and all of $f_1, ..., f_n$ are declared as `mutable`, then z must be mutable in the pre-context. Because Mut Exclusivity (Definition 4.11) holds in the pre-context/state, we get that $w.g_2$ would not be readable. This is not allowed by the Pseudoargument rule.

  If $x.f_1$ is readable, then z has deep readability (so $z.f_2$ is readable) and $p > 0$. With Mut Exclusivity (Definition 4.11) in the pre-context/state we get that $w.g_2$ cannot be mutable or one of $g_2, ..., g_m$ is not declared as `mutable`. Since Pseudoargument requires deep readability, we get that one of $g_2, ..., g_m$ so also one of $g_1, ..., g_m$ must not be declared as `mutable`.

- v_x `= new` $C(a,b)$`;` with $x \neq y$:

  Then there must be an argument with id z for f_1: $x.f_1.\cdots.f_n \equiv z.f_2.\cdots.f_n \equiv y.g_1.\cdots.g_m$.

  The Pseudoargument rule ensure a borrow $z.base \xrightarrow{p} x$ in the post-context.

  If $x.f_1$ is mutable and all of $f_1, ..., f_n$ are declared as `mutable`, then $p = 1$ and z must have deep mutability in the pre-context. The borrow ensures that $z.f_2$ is not readable as long as x is *active*. Because Mut Exclusivity (Definition 4.11) holds in the pre-context/state, we get that $y.g_1$ cannot be readable as long as z is *active*, which, by the borrow, is as long as x is *active*.

  If $x.f_1$ is readable, then $p > 0$ and z must have deep readability in the pre-context. By the borrow, $z.f_2$ is not mutable as long as x is *active*. Because Mut Exclusivity (Definition 4.11) holds before, we get that $y.g_1$ cannot be mutable as long as z is active, so also as long as x is active.

- v_y `= new` $C(a,b)$`;` with $x \neq y$:
  This is again the case above with x, y swapped. Because the return mutability of the pseudocall is always `mutable`, we either have a deep mutable y or not all of $g_1, ..., g_m$ are mutable (which covers the second property of Mut Exclusivity (Definition 4.11)).

  If $x.f_1$ is mutable in any following context, where both x, y are *active*, we get from the previous case that $y.g_1$ cannot be readable in the post-state. This is not possible with the Pseudocall rule.

Field-Assignment

Again we have different cases of how the equivalence can be established:

- $v_x.f_1 = v_z;$ with $x.f_1 = y.g_1$:

  Then $z.f_2.\cdots.f_n \equiv z.g_2.\cdots.g_m$.

  From the created borrow $z.base \xrightarrow{p} \otimes$ we get that z is *active* in all following contexts.

  If $f_1, ..., f_n$ are all declared as `mutable`, then z must have deep mutability and from Mut Exclusivity (Definition 4.11) in the pre-context/state we get that $z.g_2$ is not readable, which conflicts with the deep mutability.

  The same holds for $g_1, ..., g_n$.

- $v_x.f_1 = v_z;$ with $x.f_1 \neq y.g_1$:

  Then $z.f_1.\cdots.f_n \equiv y.g_1.\cdots.g_m$ and there is a borrow $z.base \xrightarrow{p} \otimes$ in the post-context.

  If $x.f_1$ is mutable and all of $f_1, ..., f_n$ are declared as `mutable`, then $p = 1$ and z must have deep mutability in the pre-context. The borrow ensures that $z.f_2$ is not readable in the post-state and by Mut Exclusivity (Definition 4.11) in the pre-context/state, we get that $y.g_1$ is not readable as long as z is *active* (which, by the borrow, is forever).

  If $x.f_1$ is readable, then $p > 0$ and z must have deep readability in the pre-context. The borrow ensures that $z.f_2$ is not mutable in the post-state and by Mut Exclusivity (Definition 4.11) in the pre-context/state, we get that $y.g_1$ is not readable as long as z as long as z is *active* (which, by the borrow, is forever) or that not all of $g_1, ..., g_m$ are declared as `mutable`.

- $v_y.f_1 = v_z;$ with $x.f_1 \neq y.g_1$:
  Again, this is the above case with x, y swapped.

  The Field-Assignment and Pseudoargument rules ensure that z must have at least deep readability in the pre-context, so p for the borrow in the previous case is always > 0.
  Combined with the argument from the previous case, we get that $x.f_1$ can never be mutable or not all of $f_1, ..., f_n$ are declared as `mutable`.

  If $x.f_1$ is readable in any following context, then by the previous case, we get that $p < 1$ or z cannot have deep mutability. This is only possible, if g_1 is not declared as `mutable`.

Var-From-Field-Assignment

Again we do a case distinction over the different cases of how the equivalence can be established:

- $v_x = v_z.h;$ with $x = y$:
  Then $z.h.f_1.\cdots.f_n \equiv x.f_1.\cdots.f_n \equiv x.g_1.\cdots.g_m \equiv z.h.g_1.\cdots.g_m$ and there will be a borrow $z.h \xrightarrow{p} x$ in the post-context.

If $x.f_1$ is mutable and all of $f_1, ..., f_n$ are declared as `mutable`, then $z.h$ must be mutable. Because Mut Exclusivity (Definition 4.11) holds in the pre-context/state, we get that $z.h$ must be not readable. This is a conflict.

If $x.f_1$ is readable but not mutable, then h cannot be a `mutable` field.
If $x.f_1$ is mutable, but not all of $f_2, ..., f_n$ are declared as `mutable`, then $z.h$ must be mutable. Because Mut Exclusivity (Definition 4.11) holds in the pre-context/state, we get that $z.h$ is not mutable (conflict!) or not all of $g_1, ..., g_m$ are declared as `mutable`.

- v_x `=` v_z`.`h`;` with $x \neq y$:
  Then $z.h.f_1.\cdots.f_n \equiv x.f_1.\cdots.f_n \equiv y.g_1.\cdots.g_m$ and there is a borrow $z.h \xrightarrow{p} x$ in the post-context.

  If $x.f_1$ is mutable and all of $f_1, ..., f_n$ are declared as `mutable`, then $p = 1$ and thus $z.h$ is not readable as long as x is *active*. Because Mut Exclusivity (Definition 4.11) holds in the pre-context/state, we get that $y.g_1$ is not readable as long as z is *active*, which, by the borrow is at least as long as x is *active*.

  If $x.f_1$ is readable, then $p > 1$ and thus $z.h$ is not mutable as long as x is *active*. Because Mut Exclusivity (Definition 4.11) holds in the pre-context/state, we get that $y.g_1$ is not mutable as long as z is *active* (which, by the borrow is at least as long as x is *active*) or $g_1, ..., g_m$ are not all declared as `mutable`.

- v_y `=` v_z`.`h`;` with $x \neq y$:
  We again have the previous case with swapped x, y.

  If $x.f_1$ is mutable when x and y are *active*, then, by the previous case, $y.g_1$ cannot be readable in the post-context, which is not possible with the Pseudocall rule.

  If $y.g_1$ is readable when x and y are *active*, then, by the previous case, $y.g_1$ cannot be mutable in the post-context. This is only possible, if g_1 is not a `mutable` field, or if y has *permission sum* < 1. By Corollary 4.13 (Monotone Permission Sum All Reachable), $y.g_1$ can never become mutable in any following context.

□

Corollary 4.17 Mut Exclusivity All Reachable *Mut Exclusivity* holds in all reachable states.

Proof. In the program entry, *Mut Exclusivity* trivially holds since Section 3.2.8 requires that every reference to an object is unique. Using Lemma 4.16 (Mut Exclusivity Preservation) we get that *Mut Exclusivity* holds in every reachable state. □

Theorem 4.18 Reachable Conforming Let σ with T, Γ, B be a reachable state and context combination.

Then σ conforms to T, Γ, B: $\sigma : (T, \Gamma, B)$.

Proof. We show this by contraposition:

By Corollary 4.17 (Mut Exclusivity All Reachable) and Corollary 4.13 (Monotone Permission Sum All Reachable) we know that Mut Exclusivity (Definition 4.11) and Monotone Permission Sum (Definition 4.10) hold in every reachable state/context combination.

Assume that σ is reachable with T, Γ, B but not *conforming*. Then there must be a live variable/parameter v where $v.f$ is readable and a live variable/parameter w that references the same object as v where $w.f$ is assignable.

It follows that v and w are *active*, $\mathrm{id}_\Gamma(w) \equiv \mathrm{id}_\Gamma(v)$ and $\mathrm{frac}_\Gamma(w) = 1$. From Mut Exclusivity (4.11), we get that $\mathrm{id}_\Gamma(w) = \mathrm{id}_\Gamma(v)$.

From Lemma 4.12 (Monotone Permission Sum Preservation) we then get

$$\begin{aligned} & \text{perm-sum}_\Gamma(\mathrm{id}_\Gamma(w)) \\ \leq\ & \mathrm{frac}_\Gamma(v) + \mathrm{frac}_\Gamma(w) \\ =\ & \mathrm{frac}_\Gamma(v) + 1 \\ \leq\ & 1 \\ \\ \Longrightarrow\ & \mathrm{frac}_\Gamma(v) = 0 \end{aligned}$$

which contradicts our assumption that $v.f$ is readable. □

```java
public class Example {
  @Mutable Object x;
  @Immutable Object y;

  @Mutable Object getX(@Mutable @Scope("x") Example this) {
    return x;
  }

  @Immutable Object getY(@Immutable @Scope("y") Example this) {
    return y;
  }

  public static void main(String[] args) {
    @Mutable Example e = new Example();
    Object x = e.getX();
    Object y = e.y;
    Object y2 = e.getY();

    System.out.println(x);
    System.out.println(y);
    System.out.println(y2);
  }
}
```

Listing 5.1: Example programm typed by *borro*ℚ

5 Implementation

The type system formalized in this work is implemented in *borro*ℚ[3], a pluggable type checker for Java [11]. *borro*ℚ is implemented as an annotation processor using the EISOP fork of the Checker Framework [7], [19], a framework that simplifies building custom type checkers for Java programs.

[3] *borro*ℚ is available publicly online: https://github.com/Mr-Pine/borroQ/tree/df09fd9dcae133adf95f466a8424b0911b1c817e

The mutability and scope modifiers from our formalization translate to Java as annotations [11, Section 9.7]. An example of this in a program that successfully type-checks with *borro*ℚ can be seen in Listing 5.1.
Since our formalization language is very restricted, we need to adapt the type system to make it applicable to Java. We describe the extensions that *borro*ℚ uses in subsection 5.3.

5.1 Implementation Details

The Checker Framework [7] provides a class called `BaseTypeChecker` which is designed to be extended by most type checkers. The `BaseTypeChecker` and its associated classes make a few assumptions for the type system to facilitate an easy implementation of new checkers in a few lines of code. Some of these assumptions do not hold in our type system, most notably, assignments do not transfer types and the annotations in the program (`@Mutable` and `@Immutable`) are not the types of variables. While these incompatibilities can be worked around (the uniqueness type system described in the thesis of J. Bachmeier [2] extends the `BaseTypeChecker` and also doesn't transfer types through assignments), we decided to use the lower-level `SourceChecker` in our implementation.
Because many of the utility classes in the Checker Framework depend on the `BaseTypeChecker`, this caused some additional implementation effort as we had to reimplement the functionality in these classes. This includes the handling of stub[4] files for library methods, as that is tightly woven in with the `AnnotatedTypeFactory`. Our implementation can be kept relatively small since we only have to support the cases that are relevant for our implementation. The `SourceChecker` also doesn't check that arguments of method overrides are contravariant.
Our implementation also copies code from the Checker Framework for analyzing liveness of variables, since the class included in the Checker Framework only serves as an example and the members' visibilities don't support external use. This liveness checker also included a major bug where variable occurrences as a method receiver or on the left-hand side of assignments were not counted as a use[5].
We further encountered some minor issues in the Checker Framework that we fixed and contributed to upstream[6].

The main logic of the type system is implemented in `BorroQTransfer`, which implements the `TransferFunction` provided by the Checker Framework. The `visit*` methods in this class correspond to the type rules in our implementation (modulo the extensions described in subsection 5.3). When our implementation encounters a node without explicit support,

[4] Stub files are used to add annotations to the signatures of elements outside of the project's source code.
[5] The fixes for these were upstreamed in PR #1458 and PR #1587
[6] PR #1446, PR #1451, PR #1459, PR #1460

the default visitor emits an error[7], so that our checker cannot silently accept any programs that violate the invariants in unsupported features.
Type rules that work at any stage in the program (e.g. Recombination, Null-Borrow-Deletion) are applied after every statement. This ensures that permission is returned as soon as possible.

5.2 Defaults

To simplify the use of our checker, we provide defaults when annotations are not placed on elements. Almost all locations where a mutability can be specified default to being immutable (`@Immutable`). The only exception is the mutability of the reference returned from a constructor, which is mutable by default.
Parameter scopes include *base* and all fields of the type by default.

5.3 Extensions of the Type System for Java

To make our checker usable on real code, we extend the type rules used by our type system with some utility features. These extensions can be disabled in the checker, resulting in type errors when they would be necessary.
Allow all Java primitives and `void`:
We treat all primitives identical to the boolean values in our formalization.

Support constructors:
The formalization of our language does not allow constructors with any logic. To support them in our implementation, we treat constructor calls as normal functions that start out with a mutable `this` with full permission.

Allow arbitrary number of fields in classes and parameters on functions:
To simplify the formalization, our language only allows exactly two fields on classes and exactly two parameters on functions. By extending Pseudocall to support any argument count and extending scopes to support all fields, this restriction can easily be lifted.

Generic Type Parameters:
We support generic type parameters. When type checking an argument where the parameter's type is a type parameter, we use the mutability annotation on the method's parameter. For a generic return type, we treat the return type as immutable, if the method's return type or the type parameter are immutable.

[7] This is configurable to be just a warning or suppressed

Type checking for arrays:
Our implementation supports arrays by translating them to the following pseudocalls:

Array operation	Pseudocall
`new T[]{elems...}`	`@Mutable T[] psedocall(T elem1, ...)`
`arr[idx]`	`T psedocall(<component mutability> T[] arr, int idx)`
`arr[idx] = e`	`void pseudocall(@Mutable T[] arr, <component mut> T e)`

where the component type behaves identical to a generic type argument.

More flexible arguments:
We extend the formalization to allow more than simple local variables as arguments to functions. This includes field accesses of arbitrary depth, method calls and array accesses. We achieve this by introducing *free permissions* that represent a fractional permission without an assigned id. These free permissions carry free borrows, which are borrows that specify whether their target is $\otimes$ or an id, but not which id.
When a value with a free permission is assigned to a variable, we create a new id and assign a permission with that id and the fraction of the free permission. Free borrows that need an id are filled out with the id and all borrows are inserted into the borrow list. If a standalone statement has a free permission, we insert the free borrows with $\otimes$ into the borrow list.
When using a value with a free permission as an argument, we transfer the free borrows of the value to the produced free permission of the pseudocall.

Disallow assigning out-of-scope fields:
Our formalization allows assigning fields that are out-of-scope (Section 3.1.3). Because we deem this unintuitive, *borro*$\mathbb{Q}$ does not allow this. This is achieved by checking whether the id of the receiver of a field assignment is created from a parameter. We can then check, whether the assigned field is in scope.

Allow more control-flow structures:
Thanks to the data flow analysis framework included in the Checker Framework, support for all control-flow structures is achieved by defining a function to merge two type contexts (which is defined in our formalization by the If-Statement rule).
In static analysis, merging of contexts often requires a "widening" fallback to ensure convergence of loops. Without this, our system's borrowed fractions might not converge for loops. However, we can use one of our type systems properties to avoid the widening and thus retain more precise results: The precise fraction p of a borrow is irrelevant if $0 < p < 1$. This allows us to merge two borrows with fractions less than one to the minimum of their fraction instead of their maximum, ensuring convergence without the need to widen the fraction to 1.

5.4 Interfaces and Visibility with Scopes

Since scopes can only widen a type, they are fundamentally compatible with inheritance and interfaces. We do however, especially with interfaces, lose the benefit of the fine-grainedness that these scopes give us if we only have the interface/superclass information at the call site.
A related problem is that scopes expose implementation details of the methods. If we scope a parameter, we have to mention each used field, including private ones.
These problems also appear in the theorized View Types for Rust [17] and the proposed solution is also applicable to our type system. By introducing public "field groups", the internal details can be hidden behind them, and by declaring the groups in interfaces, we could gain the granularity even when using interfaces. A similar idea, called "locset", is used for the Java Modeling Language [8, 5.9].

6 Evaluation

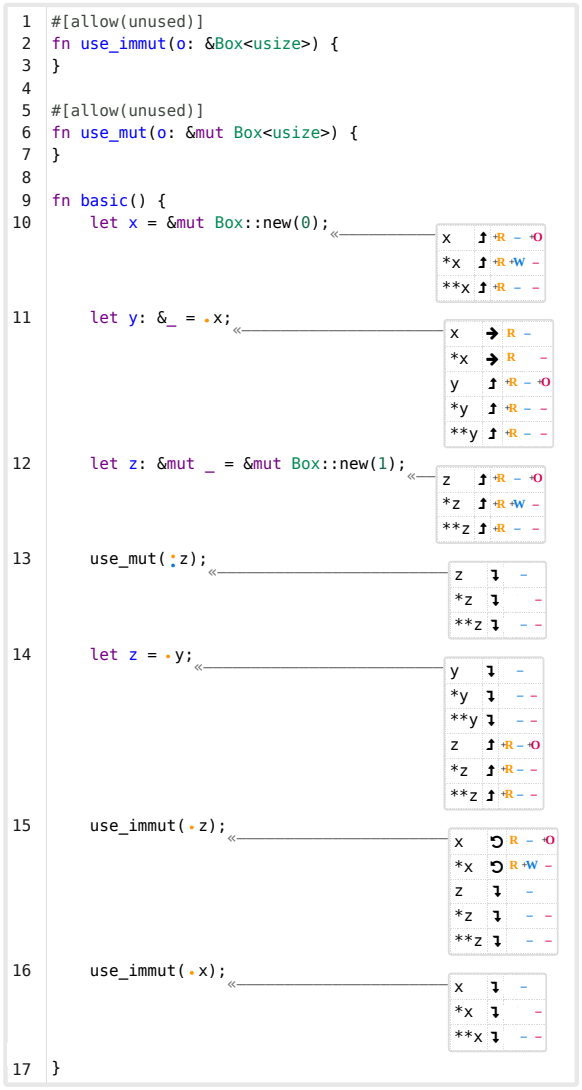
To evaluate our type system and its implementation in *borro*ℚ, we take a look at a few small code examples and compare *borro*ℚ's support for them with Kukicha's existing Uniqueness Checker [16] and Rust's ownership and borrow checker [21].

6.1 Basic example

For the first evaluation case we examine is a basic example that is supported by all three systems we compare. Listing 6.1 shows implementations of this example in Java with *borro*ℚ annotations, as well as the corresponding code in Rust.
For *borro*ℚ, the code is annotated with the corresponding contexts. The Rust code is annotated with the internal permissions the compiler uses to borrow check the program, which were extracted using the *Aquascope* tool [9].

```
1  static void basic() {                    Java
2    @Mutable Object x = new
     Object();
3    // x: [1/1]_x
4    @Immutable Object y = x;
5    // x: [1/2]_x, y: [1/2]_x
6    @Mutable Object z = new
     Object();
7    // ..., z = [1/1]_z
8    useMut(z);
9    // ...
10   // z.base —1→ ⊗
11   z = y;
12   // x: [1/2]_x, y: [1/4]_x
13   // z: [1/4]_x
14   // ———— Recombination
15   // x: [3/4]_x, y: [0]_x
16   // z: [1/4]_x
17   use(z);
18   // ..., x.base —1/8→ ⊗
19   use(x);
20   // x: [3/4]_x, y: [0]_x
21   // z: [1/4]_x
22   // z.base —1→ ⊗
23   // x.base —1/8→ ⊗
24   // x.base —7/16→ ⊗
25 }
```

(a) *borro*ℚ version



(b) Rust version with permission visualizations from Aquascope [9]

Listing 6.1: Basic Example showcasing aliasing between local variables, and function calls.

6.2 Field Assignments

In the `fieldAssignment` example, we assign fields on a duplicated reference.
This showcases how *borro*ℚ's fractional permissions allow the type system to transfer full permission to `c2` in Listing 6.2-10 and return it through recombination when `c2` is not live anymore in Listing 6.2-12.
In Rust, similar code is possible, if `c` is reborrowed as follows: `let c2 = &mut *c;`.
This "returning permission" is one of the main advantages *borro*ℚ has over the Uniqueness Checker, which is unable to check this code snippet.

```java
class C {
  @Mutable A a1;
  @Mutable A a2;

  void useMut(@Mutable C this) {
  }
}

static void fieldAssignment(@Mutable C c) {
  @Mutable C c2 = c;
  c2.a1.x = new Object();
  @Mutable A a = c.a2;
  a.x = new Object();
  c.useMut();
}
```

Listing 6.2: Field Assignment example with aliasing of the `c` parameter.

6.3 Field Access

Through our borrows with field granularity, we can extract multiple fields of the same object into local variables at the same time.
Kukicha's Uniqueness Checker is also able to type the Example in Listing 6.3a through its integration with the packing type system. By unpacking the receiver `a` on a field access, the fields `x` and `y` can be handled separately. In general, this is less flexible than our borrows since it does not allow operations on the receiver `a` while it is unpacked.
Rust's borrow checker is also able to differentiate the fields and calls this feature "Splitting Borrows" [23].

```java
class A {
  @Mutable Object x;
  @Immutable Object y;
}

static void fieldAccess(@Mutable A a) {
  @Mutable Object x = a.x;
  @Immutable Object y = a.y;
  useMut(x);
  use(y);
  a = new @Mutable A();
  x = a.x;
  y = a.y;
  useMutA(a);
}
```

(a) *borro*ℚ Version. Due to the function argument handling, the permission returning showcase needs a new receiver object.

```rust
struct A {
  x: Box<usize>,
  y: Box<usize>,
}

fn field_access(a: &mut A) {
  let x = &mut a.x;
  let y = &a.y;
  use_mut(x);
  use_immut(y);
  use_mut(a);
}
```

(b) Rust Version

Listing 6.3: Field Access Example. We first show that we can extract different fields into variables that are live concurrently and use them (Lines 7-10). The second part shows that permission to the fields is returned to the receiver (Lines 11 and following).

6.4 Parallel Getters

As motivated and described in Section 3.1.3, *borro*ℚ uses *scopes* to enable getters and multiple getters in particular. The goal is for *borro*ℚ to be able to handle direct field accesses

and getters in an identical way, although this is not quite possible because of the way we handle function arguments as we further evaluate in subsection 6.5.
In Listing 6.4 we showcase a pattern which we call “Parallel Getters”: We are able to get references to different fields of an object through getters that are live concurrently, even if one of the getters returns a mutable reference.
In *borro*ℚ this is possible by scoping these getters to their respective fields. This allows *borro*ℚ to create borrows only for this field, leaving all other field(s) accessible.
In Kukicha, getters that return a mutable, or in their terms, a `@Unique` reference are only possible if we mark the getter with `@EnsureUnknownInit`. This causes an unpacking of the receiver, which should prevent it from being used until all fields are reassigned, disallowing the second getter. Due to a bug in the implementation however, the second call is not rejected, allowing us to construct a counterexample that violates the uniqueness guarantees which is shown in Listing 6.5.
Rust’s borrow checker can also not support this pattern. The call to the getter in line 7 creates a borrow for the whole of `a` which causes an error in the next line. Solutions for this that are very similar to our scopes which are called *partial borrows* or *view types* have been discussed but are not implemented yet [17], [22].

Java

```java
class A {
  @Mutable Object x;
  @Immutable Object y;

  @Mutable Object getX(@Mutable @Scope("x") A this) { return x; }
  @Immutable Object getY(@Immutable @Scope("y") A this) { return y; }
}

static void parallelGetters() {
  @Mutable A a = new A();
  @Mutable Object x = a.getX();
  Object y = a.getY();
  useMut(x);
  use(y);
}
```

(a) *borroℚ* Version

Rust

```rust
impl A {
    fn get_x(&mut self) -> &mut Box<usize> { &mut self.x }
    fn get_y(&self) -> &Box<usize> { &self.y }
}

fn parallel_getters(a: &mut A) {
    let x = a.get_x();
    let y = a.get_y(); // This line errors because it cannot borrow *a
    use_mut(x);
    use_immut(y);
}
```

(b) Rust Version

Listing 6.4: Parallel Getters

```java
class A {
  @Unique Object x;
  @MaybeAliased Object y;

  @EnsuresUnknownInit(targetValue = Object.class)
  @Unique
  Object getX(@Unique A this) { return x; }
}

static void parallelGettersCounterExample() {
  class X {
    String value;

    X(String value) { this.value = value; }

    @Override
    public String toString() { return value; }
  }
  X xValue = new X("Hello");

  @Unique A a = new A();
  a.x = xValue;
  @Unique Object x1 = a.getX();
  @MaybeAliased Object x2 = a.getX();

  System.out.println(x2); // "Hello"
  ((X) x1).value = "World";
  System.out.println(x2); // "World"
}
```

Listing 6.5: Uniqueness Checker Counterexample that successfully type-checks.

```java
1    void uniquenessCheckerCounterExample() {
2      X xValue = new X("Hello");
3
4      @Mutable A a = new A();
5      a.x = xValue;
6      @Mutable Object x1 = a.getX();
7      // :: error: permission.insufficient.deep
8      @Immutable Object x2 = a.getX();
9
10     System.out.println(x2);
11     ((X) x1).value = "World";
12     System.out.println(x2);
13   }
```

Listing 6.6: *borro*ℚ version of the uniqueness checker counterexample. We get an "insufficient deep permission" error in line 8 since we have a borrow from `a.x` which is not compatible with calling `getX` on `a`.

6.5 Setters

A drawback of *borro*ℚ is the loss of permissions after function calls. The following example demonstrates this: The `setX` call with `a` as the receiver "consumes" all permission of `a`, not allowing any access afterward. With the modification that we propose for future work in Section 8.1.1, this problem could be significantly reduced.
Rust is able to avoid this problem through its ownership system. Taking the value as an owned argument allows the borrow checker to reason that the callers reference stays unique after the setter.
Kukicha's Uniqueness Checker solves this by requiring `@EnsuresReadOnly` or `@EnsuresMaybeAliased` annotations if an argument's permissions are different at the end of a method. This is similar to what we propose in Section 8.1.2, although their implementation can be much simpler since the Uniqueness Checker cannot return permissions.

```java
class A {
    @Mutable Object x;
    @Immutable Object y;

    void setX(@Mutable @Scope("x") A this, @Mutable Object x) {
        this.x = x;
    }
}

static void setters(@Mutable A a) {
    a.setX(new Object());
    // :: error: permission.insufficient.deep
    useMutA(a);
}
```

(a) *borro*ℚ Version

```rust
impl A {
    fn set_x(&mut self, v: Box<usize>) { self.x = v; }
}

fn setters(a: &mut A) {
    let new_x = Box::new(2);
    a.set_x(new_x);
    use_mut(a);
}
```

(b) Rust Version

Listing 6.7: Setters Example

6.6 Arrays

Both *borro*ℚ and Rust have support for arrays. The Uniqueness Checker does not, but, differing from *borro*ℚ's handling of unsupported features, it does not report this and silently accepts programs with arrays, even if they violate uniqueness.
Mirroring the feature from Rust, *borro*ℚ also supports concurrent access to different fields on array elements.

```java
1 static void arrays(@Mutable A a1, @Mutable A a2, int i, int j)
  {
2     @Mutable A[] arr = new @Mutable A[]{a1, a2};
3     @Mutable Object x = arr[i].x;
4     Object y = arr[j].y;
5     useMut(x);
6     use(y);
7 }
```

(a) *borro*ℚ Version

```rust
1 fn arrays(a1: A, a2: A, i: usize, j: usize) {
2     let mut arr = [a1, a2];
3     let x = &mut arr[i].x;
4     let y = &arr[j].y;
5
6     use_mut(x);
7     use_immut(y);
8 }
```

(b) Rust Version

Listing 6.8: Array Support: `arr[i]` and `arr[j]` in lines 3 and 4 potentially reference the same object but we can still allow the local variables `x` and `y` to be alive concurrently since they reference different fields. This guarantees that they do not alias each other.

6.7 Handling Function Arguments

The following example demonstrates the justification for *borro*ℚ behavior on function arguments. We need to "consume" all permission of passed arguments, since they could "escape" the function through other arguments.
In this case, the argument `a` could be accessed mutably through `b.a`, which would violate our invariants, if we returned permissions.
Rust avoids this problem through its ownership and lifetime model.

```java
static void func(@Mutable A a, @Mutable B b, Object y) {
    a.y = y;
    b.a = a;
}

static void callFunc(@Mutable A a, @Mutable B b) {
    func(a, b, new Object());
    // :: error: permission.insufficient.shallow
    useA(a);
}
```

Listing 6.9: Leaking Function Arguments Example. The argument `a` leaves `func` through `b` so we cannot give back permission at the call-site.

6.8 Error Messages

Rust's error messages for borrow checker violations are generally regarded as very good [3]. The error messages explain what permission is missing, where the borrow or move that resulted in the lower permission is located and, if possible, how to fix the error.
*borro*ℚ reports errors with the location where they originate along with the expression that has not enough permissions and the permission that is required.
In the uniqueness checker, some of the error messages are reported non-locally, missing information on where the error originated. This restriction is explained in the thesis by J. Bachmeier [2, 5.1] introducing the uniqueness checker.

6.9 Realistic Example

To explore *borro*ℚ's ability to type-check more realistic examples, we decided to apply it to an existing[8] solution for Advent of Code 2025 Day 1[9]. With some slight refactoring to work around *borro*ℚ's restrictions, including extracting `String` literal arguments into local variables, we were able to correctly type-check the program. This demonstrates that *borro*ℚ is expressive enough to also type more complex programs, including `try/catch` structures and `while` loops.

[8] https://github.com/ShawnWebDev/AdventOfCode25/blob/master/src/main/java/com/webdev/day1/Day1Modulo.java

[9] https://adventofcode.com/2025/day/1

```java
static int runDay1() {
  int zeroCount = 0;
  @Immutable String filename = "src/main/resources/day1_Input.txt";
  @Immutable File input = new File(filename);
  int dialPos = 50;

  try {
    @Mutable Scanner sc = new Scanner(input);

    while (sc.hasNextLine()) {
        String line = sc.nextLine();
        char firstChar = line.charAt(0);
        String firstCharString = String.valueOf(firstChar);
        String leftPrefix = "L";
        int direction = firstCharString.equals(leftPrefix) ? -1 : 1;
        String amountSubstring = line.substring(1);
        int amount = Integer.parseInt(amountSubstring);

        for (int i = 0; i < amount; i++) {
            dialPos = (dialPos + direction) % 100;
            if (dialPos == 0) { zeroCount++; }
        }
    }
    sc.close();
  } catch (FileNotFoundException e){
    String errorMessage = "File not found";
    System.out.println(errorMessage);
  }

  return zeroCount;
}
```

Listing 6.10: Advent Of Code Solution

6.10 Summary

The evaluation shows that *borro*ℚ offers improvements on the Uniqueness Checker in multiple areas. By enabling the return of permissions through fractional permissions and the concept of scope, we achieve a greater expressiveness. By always reporting errors at

their origin, we achieve better error messages, although we are not on par with Rust. *borro*ℚ's main drawback is its inflexible handling of function arguments which in some cases prohibits our type-system to type code that is accepted by the Uniqueness Checker. Scopes enable support for getters which are very common in real-world Java code, which is not possible with Rust's borrow checker.
By successfully typing an existing Advent Of Code Solution, we show that *borro*ℚ is able to handle real programs.

7 Related Work

In their thesis introducing Kukicha's Uniqueness Checker, J. Bachmeier [2, Section 5.] already gives a comprehensive overview over works that can be used to solve our problem. They present different techniques that can be used for limiting aliasing and mutation and discuss how they are suited for ensuring that mutation can only occur on exclusive references.
Since our work addresses the same problem as the Uniqueness Checker, we focus our discussion on works related to our proposed solution.

7.1 Fractional Permissions

Fractional permissions as a concept were introduced by J. Boyland [4] in 2003 to enable deterministic parallelism. Through the concept of recombining two fractional permissions, they enable regaining write permissions once all read-only references are not live anymore.
In their paper *Fractional Permissions without the Fractions*, S. Heule *et al.* [13] argue that exact fractions are not required in the specification. Specifying a concrete fractional permission for arguments makes the function harder to specify and less flexible, because it would require the exact permission from the caller. Their work shows that conceptual annotations at the level of read and write permissions suffice, and concrete values are only required for the correctness proof. We use this result and only let the user specify arguments as mutable or immutable, making annotating code much easier.
With ConSORT, J. Toman *et al.* [24] introduce a system that ensures the safety of their refinement type system using fractional permissions. They formalize their type system and provide an implementation for their own (extended) IMP language. While not supporting fields, ConSORT can handle tuple types but without access to single values of a tuple.

7.2 Typing Field Permission

J. Boyland [6] extend their *fractional permissions* by the concept of "nesting" which can model permissions of fields. In the type system we present, there is the concept of borrows instead, but borrowing can be seen as analogous to the "carving" operations described in J. Boyland's work.

Our handling of fields is inspired by Rust's reference and borrowing system [21] and its handling of borrowing different fields concurrently [23]. There has been work on systems including *Oxide* [26] and *RustBelt* [15] that try to formalize Rust's borrow checker. *Stacked Borrows* [14] and the related *Tree Borrows* [25] formalize a set of invariants that Rust code has to adhere to. They do this in addition to the borrow checker with the goal of enabling optimizations in the presence of `unsafe` code by letting the compiler assume these invariants. The graphs our borrows form are very similar to the stacks and trees in these works, although they are represented differently and follow different type rules.
For our argument scopes, the theorized *View Types* [17] and *Partial Borrowing* [22] extensions to Rusts type system served as our main inspiration. While their syntax is still under discussion, they represent the same idea of restricting a type to a subset of its fields. They both introduce the idea of declaring (publicly visible) field groups to abstract internal behavior.
When adapting concepts from Rust to our formalization language or Java, one has to adapt these concepts to the fundamental differences between these languages. The main one is that Rust differentiates between references and owned values. These explicit references and associated lifetimes are essential to some parts of Rust's borrow checking system, but do not exist in Java. There is prior work to adapt Rust's borrow checker to Java and specifically the Checker Framework in S. Harrap's *Affine Pointer Analysis for Checker* project. This checker suffers from some restrictions, including not being able to type-check field interactions, and does not provide a formalization of their type system.

8 Conclusion

In this thesis, we developed a type system for imperative languages that is able to ensure that mutation only occurs on unique references. We gave an explanation of the type rules and its design decisions as well as a formal description.
By providing a formal proof, we showed that our system does in fact guarantee uniqueness for every mutation.
As part of our work, we also developed *borro*ℚ, a type checker using the Checker Framework that implements the type rules we formalized and adapts them to Java.
This type system is able to type multiple programs that cannot be handled by the existing Uniqueness Checker in Kukicha. By our use of scopes, we are able to type a pattern we call *parallel getters* which cannot be handled by either the Uniqueness Checker or Rust's borrow checker.

8.1 Future Work

The main restriction of our type system and *borro*ℚ is the handling of function arguments as shown in the example in subsection 6.5. We propose two ideas to lift these restrictions that can be explored in future work.
To reiterate, all function arguments are borrowed to ⊗. This is necessary, because function arguments could leave the function through other mutable arguments or the return value. If we returned permission to the arguments, we could thus violatate our invariants (compare Section 3.1.2.4 and subsection 6.7).

8.1.1 Eliminating Persistent Borrows

We notice that arguments can only leave a function through mutable arguments or the return value. With this information, it is possible to avoid borrowing to ⊗ if we instead borrow to all mutable arguments and the return value's new id. If there is more than one mutable argument or a mutable argument and a non-void return value, this is equivalent to borrowing to ⊗ as it would introduce a cyclic chain of borrows, preventing them from going inactive. In the other cases, each value would be borrowed to the only argument/return value through which it could leave the function.

This is a relatively minor change to the type system but would require a change to the soundness proof.

8.1.2 Getting Additional Information

A more complicated way to solve the problem is to add more information to variables. If we can ensure that an argument value can leave the function only through the original argument, there would be no need for a borrow from this argument. This could be achieved by limiting wheter an argument's value can be assigned to a field of an argument and/or returned value.
This information has to be tracked throughout the function, increasing the complexity of the type system. In return, we gain more information about what happens with the arguments inside the function, allowing for more precise borrows.

8.1.3 Kukicha Integration

While we motivate our work by the need of Kukicha [16] for a more expressive exclusivity type system, both our formalization and *borro*ℚ are designed to be standalone.
Kukicha's requirements are more relaxed than what our type system offers. An exclusivity type system for Kukicha needs to ensure that no property type of an alias is violated by a mutation [16, Section 5].
For use with Kukicha, our type system can thus be relaxed if we restrict the ability of variables to have property types. For example, we can always allow immutable aliases that do not have a property type. If we forbid property types for all aliases of a reference, we can even allow multiple mutable aliases, as it guarantees no interaction with property types.

Bibliography

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification – The KeY Book*, vol. 10001. in Lecture Notes in Computer Science, vol. 10001. Cham: Springer International Publishing, 2016. doi: 10.1007/978-3-319-49812-6.

[2] J. Bachmeier, "Property Types for Mutable Data Structures in Java," 2022. doi: 10.5445/IR/1000150318.

[3] J. Beránek, "Evolution of Rust Compiler Errors." Accessed: Feb. 27, 2026. [Online]. Available: https://kobzol.github.io/rust/rustc/2025/05/16/evolution-of-rustc-errors.html

[4] J. Boyland, "Checking Interference with Fractional Permissions," *Static Analysis*, vol. 2694. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 55–72, 2003. doi: 10.1007/3-540-44898-5_4.

[5] J. Boyland, "Why We Should Not Add Readonly to Java (Yet).," *J. Object Technol.*, vol. 5, no. 5, pp. 5–29, 2006, Accessed: Feb. 07, 2026. [Online]. Available: http://www.cs.ru.nl/E.Poll/ftfjp/2005/Boyland.pdf

[6] J. Boyland, "Semantics of Fractional Permissions with Nesting," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 22:1–22:33, Aug. 2010, doi: 10.1145/1749608.1749611.

[7] Checker Framework Developers, "The EISOP Checker Framework." Accessed: Feb. 16, 2026. [Online]. Available: https://eisop.github.io/cf/

[8] D. R. Cok, Gary T. Leavens, and M. Ulbrich, *Java Modeling Language (JML) Reference Manual*, 2nd ed. 2026.

[9] W. Crichton, G. Gray, and S. Krishnamurthi, "A Grounded Conceptual Model for Ownership Types in Rust," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 265:1224–265:1252, Oct. 2023, doi: 10.1145/3622841.

[10] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy, "Uniqueness and Reference Immutability for Safe Parallelism," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, in OOPSLA '12. New York, NY, USA: Association for Computing Machinery, Oct. 2012, pp. 21–40. doi: 10.1145/2384616.2384619.

[11] J. Gosling, B. Joy, G. Steele, *et al.*, "The Java® Language Specification." Accessed: Feb. 07, 2026. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se25/html/index.html

[12] S. Harrap, "Sharrap/Affinechecker." Accessed: Mar. 01, 2026. [Online]. Available: https://github.com/sharrap/affinechecker

[13] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers, "Fractional Permissions without the Fractions," in *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*, Lancaster United Kingdom: ACM, Jul. 2011, pp. 1–6. doi: 10.1145/2076674.2076675.

[14] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked Borrows: An Aliasing Model for Rust," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 41:1–41:32, Dec. 2019, doi: 10.1145/3371109.

[15] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the Foundations of the Rust Programming Language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 66:1–66:34, Dec. 2017, doi: 10.1145/3158154.

[16] F. Lanzinger, J. Bachmeier, M. Ulbrich, and W. Dietl, "Kukicha – Efficient Yet Powerful Refinement Types for Object-Oriented Languages," 2025.

[17] N. Matsakis, "View Types for Rust." Accessed: Nov. 17, 2025. [Online]. Available: https://smallcultfollowing.com/babysteps/blog/2021/11/05/view-types/

[18] N. Matsakis and NLL working group, "Non-Lexical Lifetimes (NLL) Fully Stable | Rust Blog." Accessed: Feb. 18, 2026. [Online]. Available: https://blog.rust-lang.org/2022/08/05/nll-by-default

[19] M. M. Papi, M. Ali, T. L. Correa, J. H. Perkins, and M. D. Ernst, "Practical Pluggable Types for Java," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, in ISSTA '08. New York, NY, USA: Association for Computing Machinery, Jul. 2008, pp. 201–212. doi: 10.1145/1390630.1390656.

[20] Rust Contributors, "Rust Programming Language." Accessed: Mar. 04, 2026. [Online]. Available: https://rust-lang.org/

[21] Rust Contributors, "References and Borrowing - The Rust Programming Language." Accessed: Aug. 18, 2025. [Online]. Available: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

[22] Rust Contributors, "Partial Borrowing (for Fun and Profit) · Issue #1215 · Rust-Lang/Rfcs." Accessed: Nov. 17, 2025. [Online]. Available: https://github.com/rust-lang/rfcs/issues/1215

[23] Rust Contributors, "Splitting Borrows - The Rustonomicon." Accessed: Aug. 18, 2025. [Online]. Available: https://doc.rust-lang.org/nomicon/borrow-splitting.html

[24] J. Toman, R. Siqi, K. Suenaga, A. Igarashi, and N. Kobayashi, "ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs," *Programming Languages and Systems*, vol. 12075. Springer International Publishing, Cham, pp. 684–714, 2020. doi: 10.1007/978-3-030-44914-8_25.

[25] N. Villani, J. Hostert, D. Dreyer, and R. Jung, "Tree Borrows," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 188:1019–188:1042, Jun. 2025, doi: 10.1145/3735592.

[26] A. Weiss, O. Gierczak, D. Patterson, and A. Ahmed, "Oxide: The Essence of Rust." Accessed: Mar. 02, 2026. [Online]. Available: http://arxiv.org/abs/1903.00982