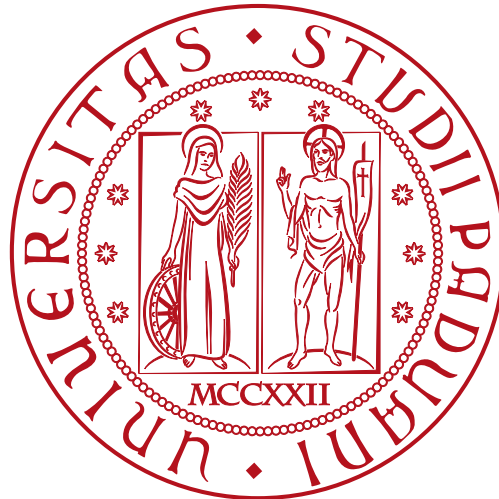# Università degli studi di Padova

DEPARTMENT OF MATHEMATICS "TULLIO LEVI-CIVITA"

MASTER'S DEGREE IN COMPUTER SCIENCE

Verifying Kotlin Code with Viper by Controlling Aliasing

Master's Thesis

Supervisor

Prof. Francesco Ranzato

Tutors

Ilya Chernikov, Komi Golov

Undergraduate

Francesco Protopapa

ID number 2079466

ACADEMIC YEAR 2023-2024

Abstract

In Computer Science, aliasing refers to the situation where two or more references point to the same object. On the one hand, aliasing can be useful in object-oriented programming, allowing programmers to implement designs involving sharing. On the other hand, aliasing poses significant challenges for formal verification. This is because changing a value through a reference can modify the data that other references also point to. As a result, it becomes more challenging to predict the behavior of the program.

Developed by JetBrains, Kotlin is an open-source, statically typed programming language that gained popularity in recent years especially in the Android software development field. However, unlike other programming languages, few tools for performing formal verification in Kotlin exist. Moreover, Kotlin does not provide any guarantee against aliasing, making formal verification a hard task for the language.

This work introduces an annotation system for a significant subset of the Kotlin language, designed to provide some formal guarantees on the uniqueness of references. After presenting and formalizing the annotation system, the thesis shows how to use these annotations for performing formal verification of Kotlin by encoding it into Viper, a language and suite of tools developed by ETH Zurich to provide an architecture for designing new verification tools. The annotation system plays a crucial role in this process, as it bridges the gap between Kotlin's lack of guarantees about aliasing and Viper's strict memory model.

"E il treno io l'ho preso e ho fatto bene
Spago sulla mia valigia non ce n'era
Solo un po' d'amore la teneva insieme
Solo un po' di rancore la teneva insieme"

—— Francesco De Gregori

Acknowledgements

Contents

x

Chapter 1

Introduction

Aliasing is a topic that has been studied for decades in computer science [8,9,16] and it refers to the situation where two or more references point to the same object. Aliasing is an important characteristic of object-oriented programming languages allowing the programmers to develope complex designs involving sharing. However, reasoning about programs written with languages that allow aliasing without any kind of control is a hard task for programmers, compilers and formal verification tools. In fact, as reported in *the Geneva Convention on the Treatment of Object Aliasing* [16], without having guarantees about aliasing it can be difficult to prove the correctness of a simple Hoare formula like the following.

$$\{x = \text{true}\} \; y := \text{false} \; \{x = \text{true}\}$$

Indeed, when x and y are aliased, the formula is not valid, and most of the time proving that aliasing cannot occur is not straightforward.

On the other hand, ensuring disjointness of the heap enables the verification of such formulas. For instance, in separation logic [19,28,29], it is possible to prove the correctness of the following formula.

$$\{(x \mapsto \text{true}) * (y \mapsto -)\} \; y := \text{false} \; \{(x \mapsto \text{true}) * (y \mapsto \text{false})\}$$

This verification is possible because separation logic allows to express that x and y are not aliased by using the separating conjunction operator "$*$". Similarly, programming languages can incorporate annotation systems [2,7,33] or built-in constructs [17,31] to provide similar guarantees regarding aliasing, thereby simplifying any verification process.

1.1 Contributions

This work demonstrates how controlling aliasing through an annotation system can enhance the formal verification process performed by SnaKt [20], an existing plugin for the Kotlin language [1,21]. SnaKt verifies Kotlin using Viper [10,25], an intermediate verification language developed by ETH Zurich. Viper is designed to verify programs by enabling the specification of functions with preconditions and postconditions, which are then checked for correctness. This verification is performed using one of two backends: symbolic execution [26] or verification condition generation [15], both of which rely on an SMT solver to validate the specified conditions.

In order to verify to Kotlin with Viper, it is necessary to translate the former language into the latter. However, this translation presents several challenges due to fundamental differences between the two languages. Specifically, Viper's memory model is based on separation logic, which disallows shared mutable references. In contrast, Kotlin does not restrict aliasing, meaning that references in Kotlin can be both shared and mutable, posing a significant challenge when trying to encode Kotlin code into Viper. This issue is clearly illustrated in the Kotlin code example provided in Listing 1. In that example, the language allows the same reference to be passed multiple times when

calling function `f`, thereby creating aliasing. Additionally, Listing 1 presents a naive approach for encoding that Kotlin code into Viper. Despite the Viper code closely resembling the original Kotlin code, it fails verification when `f(x, x)` is called. This failure occurs because `f` requires write access to the field `n` of its arguments, but as previously mentioned, Viper's separation logic disallows references from being both shared and mutable simultaneously.

```kotlin
1   class A(                    🅺 Kotlin
2       var n: Int
3   )
4
5   fun f(x: A, y: A) {
6       x.n = 1
7       y.n = 2
8   }
9
10  fun use_f(a: A) {
11      f(a, a)
12  }
```

```
1   field n: Int                🆅 Viper
2
3   method f(x: Ref, y: Ref)
4   requires acc(x.n) && acc(y.n)
5   {
6     x.n := 1
7     y.n := 2
8   }
9
10  method use_f(a: Ref)
11  requires acc(a.n)
12  {
13    f(a, a) // verification error
14  }
```

Listing 1: Kotlin code with aliasing and its problematic encoding into Viper

As mentioned before, Kotlin does not have built-in mechanisms to manage or prevent aliasing, which can lead to unintended side effects and make it harder to ensure code correctness. To address this issue, this work proposes and formalizes an annotation system specifically designed to manage and control aliasing within Kotlin.

The proposed annotation system introduces a way for developers to specify and enforce stricter aliasing rules by tagging references with appropriate annotations. This helps to clearly distinguish between references that might be shared and those that are unique. Additionally, the system differentiates between functions that create new aliases for their parameters and those that do not. This level of control is important for preventing common programming errors related to mutable shared state, such as race conditions or unintended side effects.

Listing 2 provides an overview of the annotation system. Specifically, the `@Unique` annotation ensures that a reference is not aliased, while the `@Borrowed` annotation guarantees that a function does not create new aliases for a reference. The example also demonstrates how the problematic function call presented in Listing 1 is disallowed by the annotation system, as `x` and `y` would be aliased when the function `f` requires them to be unique.

The thesis finally shows how aligning Kotlin's memory model with Viper's, using the proposed annotation system, enhances the encoding process performed by SnaKt.

```kotlin
1   class A(var n: Int)                                          ⬛ Kotlin
2
3   fun f(@Unique @Borrowed x: A, @Unique @Borrowed y: A) {
4       x.n = 1
5       y.n = 2
6   }
7
8   fun use_f(@Unique a: A) {
9       f(a, a) // annotations checking error
10  }
```

Listing 2: Kotlin code with annotations for aliasing control

1.2 Structure of the Thesis

The rest of the thesis is organized as follows:

Chapter 2 provides a description of the background information needed to understand the concepts presented by this work. In particular, this chapter presents the Kotlin programming language and its feature of interest for the thesis. Following this, the chapter provides an overview of the "Aliasing" topic in Computer Science and presents an introduction to the Viper language and its set of verification tools.

Chapter 3 analyzes works that have been fundamental for the development of this thesis. The chapter is divided in two parts, the former describing existing works about aliasing and systems for controlling it; the latter giving an overview of the already existing tools that perform formal verification using Viper.

Chapter 4 introduces a uniqueness system for the Kotlin language. It shows several examples of Kotlin code extended with uniqueness annotations and explores how the annotations can be used for bringing improvements to the language.

Chapter 5 formalizes the annotation system introduced before on a language that can represent a significant subset of the Kotlin language. After introducing the language and several auxiliary rules and functions, the typing rules for the system are formalized.

Chapter 6 shows how the annotation system presented before can be used to obtain a better encoding of Kotlin into Viper, thus improving the quality of verification performed by SnaKt.

Chapter 7 summarizes the contributions of this research and points out reasonable extensions to this work as well as potential new areas for future research.

Chapter 2

Background

This chapter outlines the background information necessary to understand the concepts discussed in the rest of this work. Specifically, it covers relevant aspects about Kotlin, aliasing, separation logic, and Viper, providing a foundation for understanding how these topics interrelate and support the main contributions of the thesis.

2.1 Kotlin

Developed by JetBrains, Kotlin [1,21] is an open-source, statically typed programming language that gained popularity in recent years, particularly in the field of Android software development. It shares many similarities with Java and it can fully interoperate with it. Additionally, Kotlin introduces a range of modern features, including improved type inference, support for functional programming, null-safety, and smart-casting, making it an attractive option for developers.
The following sections will present the features of the language that are more relevant for this work.

2.1.1 Mutability vs Immutability

In programming languages, mutability refers to the capability to alter the value of a variable after it has been initialized. In Kotlin, variables and fields can be either mutable or immutable. Mutable elements are defined using the `var` keyword, while immutable elements are defined using the `val` keyword. Mutable variables or fields, once assigned, can have their values changed during the execution of the program. In contrast, immutable elements, once assigned a value, cannot be altered subsequently. For instance, `var x = 5` allows to change the value of `x` later in the program, while `val y = 5` keeps `y` consistently at the value of `5` throughout the program's execution. This clear distinction between `val` and `var` is particularly useful in a multithreaded environment since it helps to prevent race conditions and data inconsistencies.

2.1.2 Smart Casts

In Kotlin, when the compiler can determine that a variable's type is more specific than its declared type, it inserts a smart cast to reflect this. This means that after a type check in a conditional expression, the variable can be used with its more specific type within that block without additional casting. For example, after confirming a variable is of a certain type in an `if` condition, the variable can be used with that type within the `if` block without requiring an explicit cast. An example of smart casting is provided in Listing 3.

```kotlin
1  open class A()                                          ◤ Kotlin
2  class B : A() {
3      fun f() = println("B")
4  }
5
6  fun callIfIsB(a: A) {
7      if (a is B) {
8          a.f()
9  //       ^^^^^
10 //   Smart cast to B
11     }
12 }
```

<div align="center">Listing 3: Example of smart-cast in Kotlin</div>

2.1.3 Null Safety

Kotlin's type system has been designed with the goal of eliminating the danger of
null references. In many programming languages, including Java, accessing a member
of a null reference results in a null reference exception. Kotlin avoids most of these
situations because the type system distinguishes between references that can hold `null`
and those that cannot, the former are called nullable references while the latter are
called non-nullable references. Listing 4 shows how nullable references are declared by
appending a question mark to the type name and it shows that trying to assign `null`
to a non-nullable reference leads to a compilation error.

```kotlin
1  var nullableString: String?                             ◤ Kotlin
2  nullableString = "abc" // ok
3  nullableString = null // ok
4
5  var nonNullableString: String
6  nonNullableString = "abc" // ok
7  nonNullableString = null // compilation error
```

<div align="center">Listing 4: Kotlin null safety example</div>

Accessing members of nullable reference or calling a method with a nullable reference
as receiver is only allowed if the compiler can understand that the reference will never
be null when one of these actions occurs. Usually, this is done with a smart cast con-
sidering that for every type `T`, its nullable counterpart `T?` is a supertype of `T`.

```kotlin
1  fun f(nullableString: String?) {                        ◤ Kotlin
2      if (nullableString != null) {
3          // 'nullableString' is smart-cast from 'String?' to 'String'
4          println(nullableString.length) // safe
5          println(nullableString.isEmpty()) // safe
6      }
7      val n = nullableString.length // compilation error
8  }
```

<div align="center">Listing 5: Kotlin smart cast to non-nullable</div>

However, there are instances in which a `NullPointerException` can be raised in Kotlin. These include explicit calls to `throw NullPointerException()`, performing unsafe (non-smart) casts, and during Java interoperation.

2.1.4 Properties

As mentioned before, properties in Kotlin can be declared as either mutable or read-only. While the initializer, getter, and setter for a property are optional, the property's type can also be omitted if it can be inferred from the initializer or the getter's return type. Kotlin does not allow direct declaration of fields. Instead, fields are implicitly created as part of properties to store their values in memory. When a property requires a backing field, Kotlin automatically provides one. This backing field can be accessed within the property's accessors using the `field` identifier. A backing field is generated under two conditions: if the property relies on the default implementation of at least one accessor, or if a custom accessor explicitly references the backing field via the `field` identifier.

```kotlin
class Square {                                              ⊻ Kotlin
    var width = 1 // initializer
        set(value) { // setter
            if (value > 0) field = value // accessing backing field
            else throw IllegalArgumentException(
                "Square width must be greater than 0"
            )
        }
    val area
        get() = width * width // getter
}
```

Listing 6: Kotlin properties

2.1.5 Contracts

Kotlin contracts [30] are an experimental feature introduced in Kotlin 1.3 designed to provide additional guarantees about code behavior, helping the compiler in performing more precise analysis and optimizations. Contracts are defined using a special contract block within a function, describing the relationship between input parameters and the function's effects. This can include conditions such as whether a lambda is invoked or if a function returns under certain conditions. By specifying these relationships, contracts provide guarantees to the caller of a function, offering the compiler additional information that enable more advanced code analysis.

It is important to point out that currently contracts are only partially verified by the compiler. In certain cases, the compiler trusts the contracts without verification, placing the responsibility on the programmer to ensure that the contracts are correct. In Listing 7 it is possible to see how contracts allow the initialization of immutable variables within the body of a lambda, doing this is not possible without using a contract (Listing 8).

```kotlin
1   public inline fun <R> run(block: () -> R): R {              Kotlin
2       contract {
3           callsInPlace(block, InvocationKind.EXACTLY_ONCE)
4       }
5       return block()
6   }
7
8   fun main() {
9     val b: Boolean
10    run {
11        b = true
12    }
13    println(b)
14  }
```

Listing 7: Example of contract declaration and usage

```kotlin
1   fun <R> runWithoutContract(block: () -> R): R {              Kotlin
2       return block()
3   }
4
5   fun main() {
6       val b: Boolean
7       runWithoutContract { b = true }
8   /*                      ^^^^^^^^
9       Captured values initialization is forbidden
10      due to possible reassignment
11  */
12  }
```

Listing 8: Compilation error caused by the absence of contracts

2.1.6 Annotations

Annotations provide a way to associate metadata with the code. To declare annotations, the `annotation` modifier should be placed before a class declaration. It is also possible to specify additional attributes by using meta-annotations on the annotation class. For instance, `@Target` specifies the types of elements that can be annotated. Listing 9 illustrates how to declare and use a custom annotation (Lines 1-13) alongside existing annotations such as `@Deprecated` and `@SinceKotlin`.

```kotlin
1   @Target(                                              Kotlin
2       AnnotationTarget.CLASS,
3       AnnotationTarget.FUNCTION,
4       AnnotationTarget.VALUE_PARAMETER
5   )
6   annotation class MyAnnotation
7
8   @MyAnnotation
9   class MyClass {
10      @MyAnnotation
11      fun myFun(@MyAnnotation foo: Int) {
12      }
13  }
14
15  @Deprecated(
16      message = "Use newFunction() instead",
17      replaceWith = ReplaceWith("newFunction()"),
18  )
19  fun oldFunction() { /* ... */ }
20
21  @SinceKotlin(version = "1.3")
22  fun newFunction() { /* ... */ }
```

Listing 9: Example of annotations usage

2.2 Aliasing and Uniqueness

Aliasing refers to the situation where a data location in memory can be accessed through different symbolic names in the program. Thus, changing the data through one name inherently leads to a change when accessed through the other name as well. This can happen due to several reasons such as pointers, references, multiple arrays pointing to the same memory location etc.

In contrast, uniqueness [14,24] ensures that a particular data location is accessible through only one symbolic name at any point in time. This means that no two variables or references point to the same memory location, thus preventing unintended side effects when data is modified. A data location that is accessible by exactly one reference is said to be unique; similarly, the reference pointing to that data location is also termed unique.

Uniqueness can be particularly important in concurrent programming paradigms, where the goal is often to avoid mutable shared state to ensure predictability and maintainability of the code [6]. By enforcing uniqueness, programmers can guarantee that data modifications are localized and do not inadvertently affect other parts of the program, making reasoning about program behavior and correctness more straightforward.

Listing 10 shows the concept of aliasing and uniqueness practically with a Kotlin example. The function starts by declaring and initializing variable y with x, resulting in x and y being aliased. Following that, variable z is initialized with a newly-created object in the function's second line. Therefore, at this stage in the program, z can

be referred to as "unique", signifying that it is the only reference pointing to that particular object.

```kotlin
1  class T()
2
3  fun f(x: T) {
4      val y = x // 'x' and 'y' are now aliased
5      val z = T() // here 'z' is unique
6  }
```

Listing 10: Aliasing, an example

Although aliasing is essential in object-oriented programming as it allows programmers to implement designs involving sharing, as described in *the Geneva Convention on the Treatment of Object Aliasing* [16], aliasing can be a problem in both formal verification and practical programming.

The example in Listing 11 illustrates how aliasing between references can complicate the formal verification process. In the given example, a class `A` is declared with a boolean field `x`, followed by the function `f` which accepts two arguments `a1` and `a2` of type `A`. The function assigns `true` to `a1.x`, `false` to `a2.x`, and finally returns `a1.x`. Despite the function being straightforward, we cannot assert that the function will always return `true`. The reason for this uncertainty is the potential aliasing of `a1` and `a2`, as the second assignment might change the value of `a1.x` as well.

Modern programming languages frequently utilize a high degree of concurrency, which can further complicate the verification process. As shown in Listing 12, even a simpler function than its counterpart in Listing 11 does not permit to assert that it will always return `true`. In this instance, the function only takes a single argument `a` of type `A`, assigns `true` to `a.x` and eventually returns it. However, within a concurrent context there may exist another thread with access to a variable aliasing `a` that can modify `a.x` to `false` prior to the function's return, thus challenging the verification process.

Listing 13 presents a contrived example to illustrate how aliasing can lead to mysterious bugs. Function `f` takes two lists `xs` and `ys` as arguments. If both lists are not empty, the function removes the last element from each. One might assume this function will never raise an `IndexOutOfBoundsException`. However, if `xs` and `ys` are aliased and have a size of one, this exception will occur.

Moving to a more realistic example, Listing 14 shows a reasonable C++ implementation of the assignment operator overloading for a vector. Since C++ does not have built-in mechanisms to control aliasing statically, in this implementation, the assignment operator must explicitly address the possibility of aliasing between the `this` pointer and the `&other` pointer (Lines 9-11). If these two pointers are found to be identical, indicating that the object is being assigned to itself, the operation is immediately terminated to prevent any unnecessary operations. Failing to properly manage this aliasing could lead to significant issues, such as data corruption or unintended behavior, because the operator might inadvertently delete the data before copying, thereby causing the object to lose its original state.

```kotlin
1  class A(var x: Boolean)                                    Kotlin
2
3  fun f(a1: A, a2: A): Boolean {
4    a1.x = true
5    a2.x = false
6    return a1.x
7  }
```

Listing 11: Problems caused by aliasing in formal verification

```kotlin
1  class A(var x: Boolean)                                    Kotlin
2
3  fun f(a: A): Boolean {
4    a.x = true
5    return a.x
6  }
```

Listing 12: Problems caused by aliasing in formal verification within a concurrent context

```kotlin
1   fun f(xs: MutableList<Int>, ys: MutableList<Int>) {       Kotlin
2       if (xs.isNotEmpty() && ys.isNotEmpty()) {
3           xs.removeLast()
4           ys.removeLast()
5       }
6   }
7
8   fun main() {
9       val xs = mutableListOf(1)
10      f(xs, xs)
11  }
```

Listing 13: Problems caused by aliasing in practical programming

```cpp
1  class Vector {                                                        ⟳ C++
2  private:
3      int* data;
4      size_t size;
5  public:
6      // other code here...
7
8      Vector& operator=(const Vector& other) {
9          if (this == &other) {
10             return *this;
11         }
12
13         delete[] data;
14         size = other.size;
15         data = new int[size];
16         std::memcpy(data, other.data, size * sizeof(int));
17         return *this;
18     }
19
20     // other code here...
21 }
```

Listing 14: Aliasing handling in vector assignment operator overloading

2.3 Separation Logic

Separation logic [19,28,29] is an extension of first-order logic that can be used to reason about low-level imperative programs that manipulate pointer data structures by integrating it in Hoare's triples. Unlike a first-order logic formula, which directly represents a truth value, a separation logic formula represents predicates on the heap. This enables separation logic to describe how memory locations are manipulated and how different locations interact with each other.

The core concept of separation logic is the separating conjunction $P * Q$, which asserts that P and Q hold for different, non-overlapping parts of the heap. For instance, if a change to a single heap cell affects P in $P * Q$, it is guaranteed that it will not impact Q. This feature eliminates the need to check for possible aliases in Q. On a broader scale, the specification $\{P\}\ C\ \{Q\}$ for a heap modification can be expanded using a rule that allows to derive $\{P * R\}\ C\ \{Q * R\}$, indicating that additional heap cells remain untouched. This enables the initial specification $\{P\}\ C\ \{Q\}$ to focus solely on the cells involved in the program's footprint.

Separation logic also includes other assertions: emp indicates that the heap is empty, $e_1 \mapsto e_2$ specifies that the heap contains a cell at address e_1 with the value e_2, and $a_1 \mathbin{-\!*} a_2$ asserts that extending the current heap with a disjoint part where a_1 holds will result in a heap where a_2 holds.

$$\langle\text{assert}\rangle \Coloneqq$$

$\mid$ emp	empty heap
$\mid$ $\langle\text{exp}\rangle \mapsto \langle\text{exp}\rangle$	singleton heap
$\mid$ $\langle\text{assert}\rangle * \langle\text{assert}\rangle$	separating conjunction
$\mid$ $\langle\text{assert}\rangle \mathbin{-\!\!*} \langle\text{assert}\rangle$	separating implication

Example 2.3.1: In separation logic, this example represents a Hoare triple, which consists of a precondition, a command, and a postcondition.

$$\{(x \mapsto \text{true}) * (y \mapsto \text{true})\} \ x \coloneqq \text{false} \ \{(x \mapsto \text{false}) * (y \mapsto \text{true})\}$$

The triple has the following meaning:

- Precondition $\{(x \mapsto \text{true}) * (y \mapsto \text{true})\}$ describes the state of the memory before the command is executed.
  ‣ $x \mapsto \text{true}$ means that reference x points to the value true.
  ‣ $y \mapsto \text{true}$ means that reference y points to the value true.
  ‣ Separating conjunction $*$ indicates that x and y point to different locations in memory.

- Command $x \coloneqq \text{false}$ is an assignment operation where the value referenced by x is updated from true to false.

- Postcondition $\{(x \mapsto \text{false}) * (y \mapsto \text{true})\}$ describes the state of the memory after the command is executed.
  ‣ After the assignment, x now points to false, while y continues to point to true, reflecting that y remains unchanged.
  ‣ Again, the separating conjunction $*$ ensures that x and y still point to distinct memory locations.

$\square$

Example 2.3.2: The following triple is derivable in separation logic:

$$\{(x \mapsto -) * ((x \mapsto 1) \mathbin{-\!\!*} P)\} \ x \coloneqq 1 \ \{P\}$$

$\square$

2.4 Viper

Viper [10,25] (Verification Infrastructure for Permission-based Reasoning) is a language and suite of tools developed by ETH Zurich designed to aid in the creation of verification tools. The Viper infrastructure (Figure 1) consists of the Viper intermediate language and two different back-ends: one that uses symbolic execution [26] and another that relies on verification condition generation [15].

The verification process with Viper follows several steps. First, a higher-level programming language is translated into Viper's intermediate language, which incorporates permission-based reasoning to manage and express ownership of memory locations, similar to separation logic.

After translation, Viper uses one of its back-ends and an SMT solver to verify the conditions expressed in the Viper language [12]. The back-ends are designed to automate the verification process as much as possible, allowing tool developers and users to

focus on the verification task itself without needing to comprehend the inner behavior of the back-ends
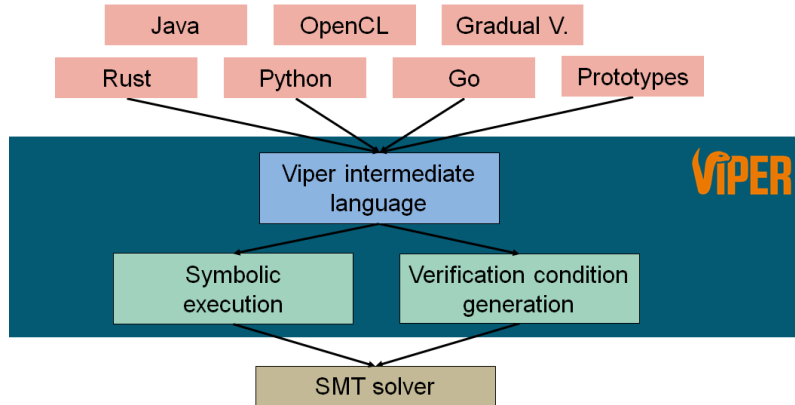
Figure 1: The Viper verification infrastructure [10]

2.4.1 Language Overview

The Viper intermediate language is a sequential, object-based language that provides simple imperative constructs along with specifications and custom statements for managing permission-based reasoning.

In Viper, methods can be seen as an abstraction over a sequence of operations. The caller of a method observes its behavior solely through the method's signature and its preconditions and postconditions. This allows Viper to perform a method-modular verification, avoiding all the complexities associated with interprocedural analysis.

```
1   method multiply(x: Int, y: Int) returns (res: Int)    ⓥ Viper
2   requires x >= 0 && y >= 0
3   ensures res == x * y
4   {
5     res := 0
6     var i: Int := 0
7     while (i < x)
8       invariant i <= x
9       invariant res == i * y
10   {
11     res := res + y
12     i := i + 1
13   }
14  }
```

Listing 15: Viper method example

Listing 15 shows an example of method in Viper. It is possible to notice that the signature of the method (Line 1) declares the returned values as a list of variables. Preconditions (Line 2), postconditions (Line 3) and invariants (Lines 8-9) are the assertions subject to verification. The remaining statements are similar to most of the existing programming languages. The language is statically typed and several built-in types like `Ref`, `Bool`, `Int`, `Seq`, `Set` and others are provided.

2.4.2 Permissions

In Viper, fields are top-level declarations and, since classes do not exist in Viper, every object has all the declared fields. Field permissions, which define the heap areas that an expression, a statement, or an assertion can access, control the reasoning of a Viper program's heap. Heap locations are only accessible if the relevant permission is under the control of the method currently being verified. Listing 16 shows how a method can require field permissions in its preconditions (Line 4) and ensure that these permissions will still be valid when returning to the caller (Line 5).

```
1  field b: Bool                                    🐍 Viper
2
3  method negate(this: Ref)
4  requires acc(this.b)
5  ensures acc(this.b)
6  {
7    this.b := !this.b
8  }
```

Listing 16: Viper permissions example

As well as being declared in preconditions and postconditions, field permissions can also be obtained within a method's body. The operation that allows to gain permissions is called inhaling and can be seen in Listing 17 (Line 3). The opposite operation is called exhaling and enables to drop permissions. Listing 17 also allows to notice how access permissions that has been seen until now are exclusive. In fact, the assertion `acc(x.b) && acc(y.b)` is similar to a separating conjunction in separation logic and so inhaling that assertion implies that `x != y`. This is confirmed by the fact that the statement at Line 6 can be verified.

```
1  field b: Bool                                    🐍 Viper
2
3  method exclusivity(x: Ref, y: Ref)
4  {
5    inhale acc(x.b) && acc(y.b)
6    assert x != y
7    x.b := true
8    y.b := true
9  }
```

Listing 17: Viper exclusivity example

Sometimes, exclusive permissions can be too restrictive. Viper also allows to have fractional permissions for heap locations that can be shared but only read. Fractional permissions are declared with a permission amount between 0 and 1 or with the `wildcard` keyword. The value represented by a `wildcard` is not constant, instead it is reselected each time an expression involving a `wildcard` is identified. The `wildcard` permission amount provides a convenient way to implement duplicable read-only resources, which is often suitable for the representation of immutable data. The example in Listing 18 shows how fractional permissions can be combined to gain full permissions (Line 6-7). In the same example it is also possible to see that Viper does not allow to have a permission amount greater than 1, in fact, since `wildcard` is an amount greater than

0, a situation in which x == y == z is not possible and so the assertion on Line 11 can be verified.

```
1    field b: Bool                                              🔺 Viper
2
3    method fractional(x: Ref, y: Ref, z: Ref)
4    requires acc(x.b, 1/2)
5    requires acc(y.b, 1/2)
6    requires acc(z.b, wildcard)
7    {
8      if (x == y) {
9        x.b := true
10       if (x == z) {
11         assert false
12       }
13     }
14   }
```

Listing 18: Viper fractional permissions example

2.4.3 Predicates and Functions

Predicates can be seen as an abstraction tool over assertions, which can include resources like field permissions. The body of a predicate is an assertion. However, predicates are not automatically inlined. In fact, in order to substitute the predicate resource with the assertions defined by its body, it is necessary to perform an unfold operation. The opposite operation is called a fold: folding a predicate substitutes the resources determined by its core content with an instance of the predicate. Having predicates that are not automatically inlined is fundamental since it allows to represent potentially unbounded data structures as shown in Listing 19 (Lines 4-8) where the predicate List can represent a linked-list. The same example shows how unfold and fold operations can be performed to access the value of the second element of a list (Lines 22-26).

Similar to predicates, functions in Viper are used to define parameterized and potentially recursive assertions. The body of a function must be an expression, ensuring that the evaluation of a function is side-effect free, just like any other Viper expression. Unlike methods, Viper reasons about functions based on their bodies, so it is not necessary to specify postconditions when the function body is provided. In Listing 19 (Lines 11-15), a function is first used to represent the size of a List, and then is utilized in the preconditions of the get_second method (Line 19).

```viper
1   field value: Int                                          🐍 Viper
2   field next: Ref
3
4   predicate List(this: Ref)
5   {
6     acc(this.value) &&
7     acc(this.next) &&
8     (this.next != null ==> List(this.next))
9   }
10
11  function size(xs: Ref): Int
12  requires List(xs)
13  {
14    unfolding List(xs) in xs.next == null ? 1 : 1 + size(xs.next)
15  }
16
17
18  method get_second(xs: Ref) returns(res: Int)
19  requires List(xs) && size(xs) > 1
20  ensures List(xs)
21  {
22    unfold List(xs)
23    unfold List(xs.next)
24    res := xs.next.value
25    fold List(xs.next)
26    fold List(xs)
27  }
```

Listing 19: Viper predicate and function example

2.4.4 Domains

Domains allow the creation of custom types, mathematical functions, and axioms that define their properties. The functions defined within a domain are accessible globally across the Viper program. These are known as domain functions, and they have more limitations compared to standard Viper functions. Domain functions cannot have preconditions and can be used in any program state. They are also always abstract, meaning that they cannot have an implemented body. To give meaning to these abstract functions, domain axioms are used. Domain axioms are also global and define properties that are assumed to be true in all states. Typically, they are expressed as standard first-order logic assertions.

```
1   domain Fraction {                                          🇷 Viper
2      function nominator(f: Fraction): Int
3      function denominator(f: Fraction): Int
4      function create_fraction(n: Int, d: Int): Fraction
5      function multiply(f1: Fraction, f2: Fraction): Fraction
6
7      axiom axConstruction {
8         forall f: Fraction, n: Int, d: Int ::
9            f == create_fraction(n, d) ==>
10              nominator(f) == n && denominator(f) == d
11     }
12
13     axiom axMultiply {
14        forall f1: Fraction, f2: Fraction, res: Fraction ::
15           res == multiply(f1, f2) ==>
16             (nominator(res) == nominator(f1) * nominator(f2)) &&
17             (denominator(res) == denominator(f1) * denominator(f2))
18     }
19  }
20
21  method m(x: Int)
22  {
23     var f: Fraction
24     f := create_fraction(x, 2)
25     assert nominator(f) == x
26     assert denominator(f) == 2
27
28     var f_sq: Fraction
29     f_sq := multiply(f, f)
30     assert nominator(f_sq) == x * x
31     assert denominator(f_sq) == 4
32  }
```

Listing 20: Viper domain example

18

Chapter 3

Related Work

This chapter first outlines the foundational principles established in *the Geneva Convention on the Treatment of Object Aliasing* [16], which serves as a fundamental reference for any work addressing aliasing issues. Then, it provides an overview of existing approaches to managing uniqueness in programming languages, focusing on the design choices that have influenced the development of the uniqueness system proposed in this work. Finally, the chapter examines current systems that utilize Viper for verification, providing a critical analysis of their strengths and limitations.

3.1 The Geneva Convention

The Geneva Convention [16] examines the issues related to aliasing management in object-oriented programming languages. After introducing the aliasing problem, the paper establishes four primary methods to manage aliasing: Detection, Advertisement, Prevention, and Control.

3.1.1 Detection

Alias detection is a retrospective process that identifies potential or actual alias patterns in a program using static or dynamic techniques. This is beneficial for compilers, static analysis tools, and programmers, as it helps detect aliasing conflicts, enables more efficient code generation, identifies cases where aliasing may invalidate predicates, and assists in resolving problematic conflicts. However, alias detection requires complex interprocedural analysis due to its non-local nature, which can make comprehensive analyses too slow to be practical. For this reason, this approach is not adopted in this work.

3.1.2 Advertisement

Given the impracticality of global detection, it is essential to create techniques and constructs that enable a more modular approach to analysis. Constructs that improve the locality of analysis by annotating methods based on their resulting aliasing behaviors can be useful for both programmers and formalists.
One example of this concept is to specify that the output of a function is not aliased anywhere else in the program, signifying that it is unique. Additionally, an "uncaptured" qualifier could state that an object is never assigned to a variable that might lead to further modifications through side channels once the method has returned.

3.1.3 Prevention

Alias prevention techniques introduce constructs that ensure aliasing does not occur in specific contexts, in a way that can be statically verified. This differs from alias advertisement, where annotations enable a modular analysis but are not checked. For static checkability, constructs must be conservatively defined. For instance, a checkable version of "uncaptured" might restrict all variable bindings within a method, except when calling other methods that also have uncaptured attributes. This approach

would forbid uses that programmers may happen to know as alias-free but cannot be statically checked to be safe.

As will be illustrated in Chapter 4 and in Chapter 5, the uniqueness system developed in this work falls into this category, as it employs conservative annotations to enforce alias prevention in a manner that can be statically verified.

3.1.4 Control

Aliasing prevention alone may not be sufficient because aliasing can be unavoidable in conventional object-oriented programming. In aliasing control, the programmer determines that the system will never reach a state where unexpected aliasing occurs, even though this possibility cannot be ruled out when examining code components individually. This is verified through an analysis of state reachability.

3.2 Systems for Controlling Aliasing

In recent decades, extensive research has been conducted to address the issue of aliasing. The book *Aliasing in Object-Oriented Programming* [8] provides a comprehensive survey of the latest techniques for managing aliasing in object-oriented programming.

3.2.1 Controlling Aliasing through Uniqueness

A uniqueness type system distinguishes values referenced no more than once from values that can be referenced multiple times in a program. Harrington's *Uniqueness Logic* [14] provides a formalization of the concept of uniqueness. While it may initially appear similar to the more widely known *Linear Logic* [13], Marshall et al. [24] clarify the differences between these two approaches and demonstrate how they can coexist. The common trait of all systems based on uniqueness is that a reference declared as unique points to an object that is not accessible by any other reference, unless such references are explicitly tracked by the system. Moreover, the unique status of a reference can be dropped at any point in the program.

A first approach to ensuring uniqueness consists of using destructive reads. Aldrich et al. [2] have developed a system called AliasJava for controlling aliasing which uses this approach. AliasJava is characterized by a strong uniqueness invariant asserting that "at a particular point in dynamic program execution, if a variable or field that refers to an object `o` is annotated unique, then no other field in the program refers to `o`, and all other local variables that refer to `o` are annotated lent". This invariant is maintained by the fact that unique references can only be read in a destructive manner, meaning that immediately after being read, the value `null` is assigned to the reference.

Boyland [7] proposes a system for controlling aliasing in Java that does not require to use destructive reads. The system utilizes a set of annotations to distinguish between different types of references. Specifically, procedure parameters and return values can be annotated as unique, indicating that they are not aliased elsewhere. Conversely, parameters and return values that are not unique are classified as shared. Within the system, a shared parameter may also be declared as borrowed, meaning that the function will not create further aliases for that parameter. Finally, fields can be marked as unique; if not, they are treated as shared. The main contribution of Boyland's work is the introduction of the "alias burying" rule: "When a unique field of an object is read, all aliases of the field are made undefined". This means that aliases of a unique field are allowed if they are assigned before being used again. The "alias burying" rule is important because it allows to avoid having destructive reads for unique references.

On the other hand, having a shared reference does not provide any guarantee on the uniqueness of that reference. Finally the object referred to by a borrowed parameter may not be returned from a procedure, assigned to a field or passed as a non-borrowed parameter.

Zimmerman et al. [33] propose an approach to reduce both the volume of annotations and the complexity of invariants necessary for reasoning about aliasing in an object-oriented language with mutation. The system requires minimal annotations from the user: fields and return types can be annotated as unique or shared, while method parameters can be marked as unique, shared, or owned. For local variables, the system automatically infers the necessary information. Furthermore, the system provides flexibility for uniqueness by permitting local variable aliasing, as long as this aliasing can be precisely determined. A uniqueness invariant is defined as follows: "a unique object is stored at most once on the heap. In addition, all usable references to a unique object from the local environment are precisely inferred". The system's analysis produces at each program point an "alias graph", that is an undirected graph whose nodes are syntactic paths and distinct paths p_1 and p_2 are connected iff p_1 and p_2 are aliased. Moreover a directed graph whose nodes are syntactic path called "reference graph" is also produced for every program point. Intuitively, having an edge from p_1 to p_2 in the reference graph means that the annotation of p_1 requires to be updated when p_2 is updated.

3.2.2 Programming Languages with Aliasing Guarantees

Recently, several programming languages have started to introduce type systems that provide strong guarantees regarding aliasing.

Rust is a modern programming language that prioritizes both high performance and static safety. A key feature of Rust is its ownership-based type system [31], which guarantees memory safety by preventing problems such as dangling pointers, data races, and unintended side effects from aliased references. The type system enforces strict rules, allowing memory to be either mutable or shared, but not both at the same time. This approach helps to avoid common memory errors and aligns Rust's memory model with principles from separation logic, facilitating formal verification [23].

Swift is another language that has introduced constructs to manage aliasing effectively [17,18]. By default, function arguments in Swift are passed by value, which means any modifications made within the function do not affect the original argument in the caller. However, parameters marked as `inout` behave differently. When a function is called with an `inout` parameter, the argument's value is copied. The function then works with this copy, and when it returns, the modified copy is assigned back to the original argument. Swift guarantees memory exclusivity, meaning that accessing an `inout` value from two different references simultaneously is prohibited, thereby preventing aliasing issues. In addition to `inout`, Swift provides two other parameter modifiers to manage ownership more precisely. The `borrowing` modifier indicates that the function temporarily accesses the parameter's value without taking ownership, leaving the caller responsible for the object's lifetime. This approach minimizes overhead when the function uses the object only transiently. Conversely, the `consuming` modifier indicates that the function takes full ownership of the value, including the responsibility for either storing or destroying it before the function returns.

Finally, Granule [27] is a language designed with a focus on fine-grained resource management. Its type system combines linear types, indexed types (lightweight dependent types), and graded modal types to enable advanced quantitative reasoning. This combination offers strong guarantees for memory management and aliasing, ensuring strict

control over when and how resources can be accessed. Granule aims to demonstrate the reasoning power of combining linear, graded, and indexed types, particularly in the context of common language features such as data types, pattern matching, and recursion.

3.3 Viper Verification Tools

Several verifiers have been built on top of Viper. The most relevant tools for this work are: Prusti, a verifier for the Rust programming language, Gobra, used to verify code written in Go, and Nagini, which can be used to verify Python programs.

All these tools require the user to add annotations to the code that has to be verified. However, the number of annotations needed is inversely proportional to the robustness of the language's type system. This is the reason why the verifier for the Rust language is able to verify significant properties even without annotations, while other verifiers cannot work without user-provided annotations.

3.3.1 Prusti

Based on the Viper infrastructure, Prusti [3,4] is an automated verifier for Rust programs. It takes advantage of Rust's robust type system to make the specification and verification processes more straightforward.

By default, Prusti ensures that a Rust program will not encounter an unrecoverable error state causing it to terminate at runtime. This includes panics caused by explicit `panic!(...)` calls as well as those from bounds-checks or integer overflows.

In addition to use Prusti to ensure that programs are free from runtime panics, developers can gradually add annotations to their code, thereby achieving increasingly robust correctness guarantees and improving the overall reliability and safety of their software.

In terms of Viper encoding, Rust structs are represented as potentially nested and recursive predicates representing unique access to a type instance. Furthermore, moves and straightforward usages of Rust's shared and mutable borrows are akin to ownership transfers within the permission semantics of separation logic assertions. Reborrowing is directly modeled using magic wands, Viper's counterpart to the separating implication in separation logic. When a reborrowed reference is returned to the caller, it includes a magic wand denoting the ownership of all locations from which borrowing occurred, except those currently in the proof.

3.3.2 Gobra

Go is a programming language that combines typical characteristics of imperative languages, like mutable heap-based data structures, with more unique elements such as structural subtyping and efficient concurrency primitives. This mix of mutable data and sophisticated concurrency constructs presents unique challenges for static program verification.

Gobra [32] is a tool designed for Go that allows modular verification of programs. It can ensure memory safety, crash resistance, absence of data races, and compliance with user-defined specifications.

Compared to Prusti, Gobra generally requires more user-provided annotations. Benchmarks by Wolf et al. [32] indicate that the annotation overhead varies from 0.3 to 3.1 lines of annotations per line of code.

3.3.3 Nagini

Nagini [11] is a verification tool for statically-typed, concurrent Python programs. Its capabilities include proving memory safety, freedom from data races, and user-defined assertions.

Programs must follow to the static, nominal type system described in PEP 484 and implemented by the Mypy type checker to be compatible with Nagini. This type system requires type annotations for function parameters and return types, while types for local variables are inferred.

The tool includes a library of specification functions to express preconditions and postconditions, loop invariants, and other assertions.

By default, Nagini verifies several safety properties, ensuring that validated programs do not emit runtime errors or undeclared exceptions. Its permission system ensures that validated code is memory safe and free of data races. Moreover, the tool can verify functional properties, input/output properties and can ensue that no thread is indefinitely blocked when acquiring a lock or joining another thread, thus including deadlock freedom and termination.

Similarly to Gobra, Nagini requires a significant amount of annotations provided by the user and requires users to write fold operations.

Chapter 4

Uniqueness in Kotlin

This chapter introduces a uniqueness system for Kotlin that takes inspiration from the systems described in Subsection 3.2.1. The following subsections provide an overview of this system, with formal rules defined in Chapter 5.

4.1 Overview

The uniqueness system introduces two annotations, as shown in Listing 21. The `Unique` annotation can be applied to class properties, as well as function receivers, parameters, and return values. In contrast, the `Borrowed` annotation can only be used on function receivers and parameters. These are the only annotations the user needs to write, annotations for local variables are inferred.

Generally, a reference annotated with `Unique` is either `null` or the sole accessible reference to an object. Conversely, if a reference is not unique, there are no guarantees about how many accessible references exist to the object. Such references are referred to as shared.

The `Borrowed` annotation is similar to the one described by Boyland [7] and also to the `Owned` annotation discussed by Zimmerman et al. [33]. In this system, every function must ensure that no additional aliases are created for parameters annotated with `Borrowed`. Moreover, a distinguishing feature of this system is that borrowed parameters can either be unique or shared.

```kotlin
1  @Target(
2      AnnotationTarget.VALUE_PARAMETER,
3      AnnotationTarget.FUNCTION,
4      AnnotationTarget.PROPERTY
5  )
6  annotation class Unique
7
8  @Target(AnnotationTarget.VALUE_PARAMETER)
9  annotation class Borrowed
```

Listing 21: Annotations for the Kotlin uniqueness system

4.1.1 Function Annotations

The system allows annotating the receiver and parameters of a function as `Unique`. It is also possible to declare that a function's return value is unique by annotating the function itself. When a receiver or parameter is annotated with `Unique`, it imposes a restriction on the caller, that must pass a unique reference, and provides a guarantee to the callee, ensuring that it has a unique reference at the begin of its execution. Conversely, a return value annotated with `Unique` guarantees to the caller that the function will return a unique object and imposes a requirement on the callee to return a unique object.

Additionally, function parameters and receivers can be annotated as `Borrowed`. This imposes a restriction on the callee, which must ensure that no further aliases are created, and guarantees to the caller that passing a unique reference will preserve its uniqueness. On the other hand, if a unique reference is passed to a function without borrowing guarantees, the variable becomes inaccessible to the caller until it is reassigned.

```kotlin
class T()

fun consumeUnique(@Unique t: T) { /* ... */ }

@Unique
fun returnUniqueError(@Unique t: T): T {
    consumeUnique(t) // uniqueness is lost
    return t // error: 'returnUniqueError' must return a unique reference
}

fun borrowUnique(@Unique @Borrowed t: T) { /* ... */ }
fun borrowShared(@Borrowed t: T) { /* ... */ }

@Unique
fun returnUniqueCorrect(@Unique t: T): T {
    borrowUnique(t) // uniqueness is preserved
    borrowShared(t) // uniqueness is preserved
    return t // ok
}

fun sharedToUnique(t: T) {
    consumeUnique(t) // error: 'consumeUnique' expects a unique argument,
but 't' is shared
}
```

Listing 22: Uniqueness annotations usage on Kotlin functions

4.1.2 Class Annotations

Classes can have their properties annotated as `Unique`. Annotations on properties define their uniqueness at the beginning of a method. However, despite the annotation, a property marked as `Unique` may still be accessible through multiple paths. For a property to be accessible through a single path, both the property and the object owning it must be annotated as `Unique`. This principle also applies recursively to nested properties, where the uniqueness of the entire chain of ownership is necessary to ensure single-path access. For example, in Listing 23, even though the property x of the class A is annotated as `Unique`, `sharedA.x` is shared because `sharedA`, the owner of property x, is shared.

Moreover, properties with primitive types do not need to be annotated. This is because, unlike objects, primitive types are copied rather than referenced, meaning that each variable holds its own independent value. Therefore, the concept of uniqueness, which is designed to manage the sharing and mutation of objects in memory, does not

apply to primitive types. They are always unique in the sense that each instance operates independently, and there is no risk of aliasing or unintended side effects through shared references.

```kotlin
1   class T()
2
3   class A(
4       @property:Unique var x: T,
5       var y: T,
6   )
7
8   fun borrowUnique(@Unique @Borrowed t: T) {}
9
10  fun f(@Unique uniqueA: A, sharedA: A) {
11      borrowUnique(uniqueA.x) // ok: both 'uniqueA' and property 'x' are unique
12      borrowUnique(uniqueA.y) // error: 'uniqueA.y' is not unique since property 'y' is shared
13      borrowUnique(sharedA.x) // error: 'sharedA.x' is not unique since 'sharedA' is shared
14  }
```

Listing 23: Uniqueness annotations usage on Kotlin classes

4.1.3 Uniqueness and Assignments

The uniqueness system handles assignments similarly to Boyland's system [7]. Specifically, once a unique reference is read, it cannot be accessed again until it has been reassigned. However, passing a reference to a function expecting a `Borrowed` argument does not count as reading, since borrowing ensures that no further aliases are created during the function's execution. This approach allows for the formulation of the following uniqueness invariant: "A unique reference is either `null` or points to an object as the only accessible reference to that object."

```kotlin
1   class T()                                                    Kotlin
2   class A(@property:Unique var t: T?)
3
4   fun borrowUnique(@Unique @Borrowed t: T?) {}
5
6   fun incorrectAssignment(@Unique a: A) {
7       val temp = a.t // 'temp' becomes unique, but 'a.t' becomes inaccessible
8       borrowUnique(a.t) // error: 'a.t' cannot be accessed
9   }
10
11  fun correctAssignment(@Unique a: A) {
12      borrowUnique(a.t) // ok, 'a.t' remains accessible
13      val temp = a.t // 'temp' becomes unique, but 'a.t' becomes inaccessible
14      borrowUnique(temp) // ok
15      a.t = null // 'a.t' is unique again
16      borrowUnique(a.t) // ok
17  }
```

Listing 24: Uniqueness behavior with assignments in Kotlin

4.2 Benefits of Uniqueness

The uniqueness annotations that have been introduced can bring several benefits to the language.

4.2.1 Formal Verification

The main goal of introducing the concept of uniqueness in Kotlin is to enable the verification of interesting functional properties. For example, it might be interesting to prove the absence of IndexOutOfBoundsException in a function. However, the lack of aliasing guarantees within a concurrent context in Kotlin can complicate such proofs [22], even for relatively simple functions like the one shown in Listing 25. In this example, the following scenario could potentially lead to an IndexOutOfBoundsException:

- The function executes xs.add(x), adding an element to the list xs.
- Concurrently, another function with access to an alias of xs invokes the clear method, emptying the list.
- Subsequently, xs[0] is called on the now-empty list, raising an IndexOutOfBoundsException.

Uniqueness, however, offers a solution by providing stronger guarantees. If xs is unique, there are no other accessible references to the same object, which simplifies proving the absence of IndexOutOfBoundsException.

```kotlin
1   fun <T> f(xs: MutableList<T>, x: T) : T {                     Kotlin
2       xs.add(x)
3       return xs[0]
4   }
```

Listing 25: Function using a mutable list

Moreover, the concept of uniqueness can significantly facilitate the process of encoding Kotlin programs into Viper. Uniqueness guarantees that a reference to an object is exclusive, meaning there are no other accessible references to it. This characteristic aligns well with Viper's notion of write access. In Viper, write access refers to a situation where a reference is guaranteed to be inaccessible to any other part of the program outside the method performing the write operation. This guarantee allows Viper to perform rigorous formal verification since it can assume that no external factors will alter the reference while it is being used.

4.2.2 Smart Casts

As introduced in Subsection 2.1.2, smart casts are an important feature in Kotlin that allow developers to avoid using explicit cast operators under certain conditions. However, the compiler can only perform a smart cast if it can guarantee that the cast will always be safe [1]. This guarantee relies on the concept of stability: a variable is considered stable if it cannot change after being checked, allowing the compiler to safely assume its type throughout a block of code. Since Kotlin allows for concurrent execution, the compiler cannot perform smart casts when dealing with mutable properties. The reason is that after checking the type of a mutable property, another function running concurrently may access the same reference and change its value. Listing 26, shows that after checking that `a.valProperty` is not `null`, the compiler can smart cast it from `Int?` to `Int`. However, the same operation is not possible for `a.varProperty` because, immediately after checking that it is not `null`, another function running concurrently might set it to `null`. Guarantees on the uniqueness of references can enable the compiler to perform more exhaustive analysis for smart casts. When a reference is unique, the uniqueness system ensures that there are no accessible aliases to that reference, meaning it is impossible for a concurrently running function to modify its value. Listing 27 shows the same example as before, but with the function parameter being unique. Since `a` is unique, it is completely safe to smart cast `a.varProperty` from `Int?` to `Int` after verifying that it is not null.

```kotlin
class A(var varProperty: Int?, val valProperty: Int?)

fun useSharedA(a: A): Int {
    return when {
        a.valProperty != null -> a.valProperty // smart cast
        a.varProperty != null -> a.varProperty // compilation error
        else -> 0
    }
}
```

Listing 26: Smart cast error caused by mutability

```kotlin
1  class A(var varProperty: Int?, val valProperty: Int?)          [K Kotlin]
2
3  fun useUniqueA(@Unique @Borrowed a: A): Int {
4      return when {
5          a.valProperty != null -> a.valProperty // smart cast to Int
6          a.varProperty != null -> a.varProperty // smart cast to Int
7          else -> 0
8      }
9  }
```

Listing 27: Smart cast enabled thanks to uniqueness

4.2.3 Optimizations

Uniqueness can also optimize functions in certain circumstances, particularly when working with data structures like lists. In the Kotlin standard library, functions that manipulate lists, such as `filter`, `map`, and `reversed`, typically create a new list to store the results of the operation. For instance, as shown in Listing 28, the `filter` function traverses the original list, selects the elements that meet the criteria, and stores these elements in a newly created list. Similarly, `map` generates a new list by applying a transformation to each element, and `reversed` produces a new list with the elements in reverse order.

While this approach ensures that the original list remains unchanged, it also incurs additional memory and processing overhead due to the creation of new lists. However, when the uniqueness of a reference to the list is guaranteed, these standard library functions could be optimized to safely manipulate the list in place. This means that instead of creating a new list, the function would modify the original list directly, significantly improving performance by reducing memory usage and execution time.

```kotlin
1  fun manipulateList(xs: List<Int>): List<Int> {          [K Kotlin]
2      return xs.filter { it % 2 == 0 }
3          .map { it + 1 }
4          .reversed()
5  }
```

Listing 28: List manipulation example

4.3 Stack Example

To conclude the overview of the uniqueness system, a more complex example is provided in Listing 29. The example shows the implementation of an alias-free stack, a common illustration in the literature for showcasing uniqueness systems in action [2,33]. It is interesting to note that having a unique receiver for the `pop` function allows to safely smart cast `this.root` from `Node?` to `Node` (Lines 19-20); this would not be allowed without uniqueness guarantees since `root` is a mutable property.

30

```kotlin
1   class Node(                                                    ⬛ Kotlin
2       @property:Unique var value: Any?,
3       @property:Unique var next: Node?,
4   )
5
6   class Stack(@property:Unique var root: Node?)
7
8   fun @receiver:Borrowed @receiver:Unique Stack.push(@Unique value: Any?) {
9       val r = this.root
10      this.root = Node(value, r)
11  }
12
13  @Unique
14  fun @receiver:Borrowed @receiver:Unique Stack.pop(): Any? {
15      val value: Any?
16      if (this.root == null) {
17          value = null
18      } else {
19          value = this.root.value
20          this.root = this.root.next
21      }
22      return value
23  }
```

Listing 29: Stack implementation with uniqueness annotations

Chapter 5

Annotation System

This chapter formalizes the uniqueness system that was introduced in Chapter 4. While inspired by prior works [2,7,33], it introduces several significant improvements. This system is designed for being as lightweight as possible and gradually integrable with already existing Kotlin code. The main goal of the system is to improve SnaKt's verification process by adding aliasing control to Kotlin, thereby establishing a connection to separation logic in Viper.

5.1 Grammar

In order to define the rules of this annotation system, a grammar representing a subset of the Kotlin language is used. This grammar captures the specific syntax and features that the system needs to handle. By focusing on a subset, the rules can be more clearly defined and easier to manage, while many complex features of the language can be supported through syntactic sugar.

$$
\begin{aligned}
P &::= \overline{\mathrm{CL}} \times \overline{M} \\
\mathrm{CL} &::= \mathsf{class}\ C\big(\overline{f : \alpha_f}\big) \\
M &::= m\big(\overline{x : \alpha_f \beta}\big) : \alpha_f \{\mathrm{begin}_m; s; \mathrm{return}_m e\} \mid m\big(\overline{x : \alpha_f \beta}\big) : \alpha_f \\
\alpha_f &::= \mathrm{unique} \mid \mathrm{shared} \\
\beta &::= \cdot \mid \flat \\
p &::= x \mid p.f \\
e &::= \mathrm{null} \mid p \mid m(\overline{p}) \\
s &::= \mathrm{var}\ x \mid p = e \mid s_1; s_2 \mid \mathrm{if}\ p_1 == p_2\ \mathrm{then}\ s_1\ \mathrm{else}\ s_2 \mid m(\overline{p})
\end{aligned}
$$

Classes are made of fields, each associated with an annotation α_f. Methods have parameters that are also associated with an annotation α_f as well as an additional annotation β, and they are further annotated with α_f for the returned value. The receiver of a method is not explicitly included in the grammar, as it can be treated as a parameter. Similarly, constructors are excluded from the grammar since they can be viewed as methods without a body returning a unique value. Overall, a program is simply made of a set of classes and a set of methods.
The annotations are the same that have been introduced in the previous chapter, the only difference is that `Borrowed` is represented using the symbol $\flat$. Finally, statements and expressions are pretty similar to Kotlin.
The runtime semantics of this grammar is not formalized in this work, as it corresponds to the expected semantics for an imperative language. Moreover, annotations do not impact the runtime behavior of the program.

```
1   class C(
2      f1: unique,
3      f2: shared
4   )
5
6   m1() : unique {
7      ...
8   }
9
10  m2(this: unique) : shared {
11     ...
12  }
13
14  m3(
15     x1: unique,
16     x2: unique ♭,
17     x3: shared,
18     x4: shared ♭
19  ) {
20     ...
21  }
```

```kotlin
class C(                          🄺 Kotlin
    @property:Unique var f1: Any,
    var f2: Any
)

@Unique fun m1(): Any {
    /* ... */
}

fun @receiver:Unique Any.m2() {
    /* ... */
}

fun m3(
    @Unique x1: Any,
    @Unique @Borrowed x2: Any,
    x3: Any,
    @Borrowed x4: Any
) {
    /* ... */
}
```

Listing 30: Comparison between the grammar and annotated Kotlin

5.2 General

$$\text{M-Type-1} \; \frac{m(x_0 : \alpha_0\beta_0, ..., x_n : \alpha_n\beta_n) : \alpha\{\text{begin}_m; s; \text{return}_m e\} \in P}{\text{m-type}(m) = \alpha_0\beta_0, ..., \alpha_n\beta_n \to \alpha}$$

$$\text{M-Type-2} \; \frac{m(x_0 : \alpha_0\beta_0, ..., x_n : \alpha_n\beta_n) : \alpha \in P}{\text{m-type}(m) = \alpha_0\beta_0, ..., \alpha_n\beta_n \to \alpha}$$

$$\text{M-Args-1} \; \frac{m(x_0 : \alpha_0\beta_0, ..., x_n : \alpha_n\beta_n) : \alpha\{\text{begin}_m; s; \text{return}_m e\} \in P}{\text{args}(m) = x_0, ..., x_n}$$

$$\text{M-Args-2} \; \frac{m(x_0 : \alpha_0\beta_0, ..., x_n : \alpha_n\beta_n) : \alpha \in P}{\text{args}(m) = x_0, ..., x_n}$$

$$\text{F-Default} \; \frac{\text{class } C\left(\overline{f' : \alpha_f'}, f : \alpha_f, \overline{f'' : \alpha_f''}\right) \in P}{\text{default}(f) = \alpha_f}$$

Given a program P, M-Type rules define a function taking a method name and returning its type. Similarly, M-Args rules define a function taking a method name and returning its arguments. In order to derive these rules, the method must be contained

within P. For simplicity, it is assumed that in P, fields within the same class, as well as across different classes, have distinct names. This assumption simplifies the definition of the F-Default rule, which defines a function that returns the type of a given field.

Example 5.2.1: Given a method:

$$m(x : \text{unique } \flat, y : \text{shared}) : \text{unique}$$

The type and the arguments of m are the following:

$$\text{m-type}(m) = \text{unique } \flat, \text{ shared} \rightarrow \text{unique}$$
$$\text{args}(m) = x, y$$

<div align="right">□</div>

5.3 Context

A context is a list of distinct paths associated with their annotations α and β. While β is defined in the same way of the grammar, α is slightly different. Other than unique and shared, in a context, an annotation α can also be $\top$. As will be better explained in the following sections, the annotation $\top$ can only be inferred, so it is not possible for the user to write it. A path annotated with $\top$ within a context is not accessible, meaning that the path needs to be re-assigned before being read. The formal meaning of the annotation $\top$ will be clearer while formalizing the statement typing rules.

$$
\begin{array}{l}
\alpha \Coloneqq \text{unique} \mid \text{shared} \mid \top \\
\beta \Coloneqq \cdot \mid \flat \\
\Delta \Coloneqq \cdot \mid p : \alpha\beta, \Delta
\end{array}
$$

Apart from $\top$, the rest of the annotations are similar to the annotations in the previous section. A reference annotated as unique may either be `null` or point to an object, with no other accessible references to that object. In contrast, a reference marked as shared can point to an object without being the only reference to it. The annotation $\flat$ (borrowed) indicates that the method receiving the reference will not create additional aliases to it, and upon returning, the fields of the object will have at least the permissions specified in the class declaration. Finally, annotations on fields only indicate the default permissions; to determine the actual permissions of a field, the context must be considered, a concept that will be formalized in the upcoming sections.

5.4 Well-Formed Context

$$\text{Not-In-Base} \frac{}{p \notin \cdot} \qquad\qquad \text{Not-In-Rec} \frac{p \neq p' \qquad p \notin \Delta}{p \notin (p' : \alpha\beta, \Delta)}$$

$$\text{Ctx-Base} \frac{}{\cdot \text{ ctx}} \qquad\qquad \text{Ctx-Rec} \frac{\Delta \text{ ctx} \qquad p \notin \Delta}{p : \alpha\beta, \Delta \text{ ctx}}$$

This first set of rules defines how a well-formed context is structured. The judgment $p \notin \Delta$ is derivable when p is not present in the context. If the judgment Δ ctx is

derivable, the context is well-formed. In order to be well-formed, a context must not contain duplicate paths and must be finite.

Example 5.4.1: Given a context:

$$\Delta = x : \text{unique}, \ x.y : \text{shared}$$

The following judgments are derivable:

$$y \notin \Delta$$
$$x.f \notin \Delta$$
$$x.y.z \notin \Delta$$

$\square$

Example 5.4.2: Given the following contexts:

$$\Delta_1 = x : \text{unique}, \ x.y : \text{shared}$$
$$\Delta_2 = x : \text{unique}, \ x.y : \text{shared}, \ x : \text{shared}$$
$$\Delta_3 = x : \text{unique}, \ x.y : \text{shared}, \ x.y : \top$$

The judgment "Δ_1 ctx" is derivable meaning that Δ_1 is a well-formed context. However, the judgments "Δ_2 ctx" and "Δ_3 ctx" are not derivable meaning that Δ_2 and Δ_3 are not well-formed contexts. Indeed, they are not well-formed because x appears twice in Δ_2 and $x.f$ appears twice in Δ_3. $\square$

5.5 Sub-Paths and Super-Paths

5.5.1 Definition

$$\text{Sub-Path-Base} \frac{}{p \sqsubset p.f} \qquad\qquad \text{Sub-Path-Rec} \frac{p \sqsubset p'}{p \sqsubset p'.f}$$

$$\text{Sub-Path-Eq-1} \frac{}{p \sqsubseteq p} \qquad\qquad \text{Sub-Path-Eq-2} \frac{p \sqsubset p'}{p \sqsubseteq p'}$$

This set of rules is used to formally define sub-paths and super-paths.

Example 5.5.1.1: Given two paths $x.y$ and $x.y.z$, the following judgment is derivable:

$$x.y \sqsubset x.y.z$$

We say that:
- $x.y$ is a sub-path of $x.y.z$
- $x.y.z$ is a super-path of $x.y$

$\square$

5.5.2 Remove

$$\text{Remove-Empty} \frac{}{\cdot \setminus p = \cdot} \qquad\qquad \text{Remove-Base} \frac{}{(p : \alpha\beta, \Delta) \setminus p = \Delta}$$

$$\text{Remove-Rec} \, \frac{\Delta \setminus p = \Delta' \quad p \neq p'}{(p' : \alpha\beta, \Delta) \setminus p = p' : \alpha\beta, \Delta'}$$

Remove rules are used to define a function taking a context and a path and returning a context.

$$\_ \setminus \_ : \Delta \to p \to \Delta$$

Basically, the function will return the context without the specified path if the path is within the context, and it will return the original context if the path is not contained.

Example 5.5.2.1: Given a context:

$$\Delta = x : \text{shared}, \ x.f : \text{shared}$$

Remove has the following results:

$$\Delta \setminus x.f = x : \text{shared}$$
$$\Delta \setminus x = x.f : \text{shared}$$
$$\Delta \setminus y = x : \text{shared}, \ x.f : \text{shared}$$

$\square$

5.5.3 Deep Remove

$$\text{Deep-Remove-Empty} \, \frac{}{\cdot \ominus p = \cdot}$$

$$\text{Deep-Remove-Discard} \, \frac{p \sqsubseteq p' \quad \Delta \ominus p = \Delta'}{(p' : \alpha\beta, \Delta) \ominus p = \Delta'}$$

$$\text{Deep-Remove-Keep} \, \frac{p \not\sqsubseteq p' \quad \Delta \ominus p = \Delta'}{(p' : \alpha\beta, \Delta) \ominus p = (p' : \alpha\beta, \Delta')}$$

Deep-Remove rules define a function similar to Remove ($\setminus$) that in addiction to removing the given path from the context, also removes all the super-paths of that path.

$$\_ \ominus \_ : \Delta \to p \to \Delta$$

Example 5.5.3.1: Given a context:

$$\Delta = x : \text{unique}, \ x.y : \text{unique}, \ x.f : \text{unique}, \ x.y.z : \text{unique}$$

Deep Remove has the following result:

$$\Delta \ominus x.y = x : \text{unique}, \ x.f : \text{unique}$$

$\square$

5.5.4 Replace

$$\text{Replace} \, \frac{\Delta \ominus p = \Delta'}{\Delta[p \mapsto \alpha\beta] = \Delta', p : \alpha\beta}$$

This rule gives the definition of a function that will be fundamental for typing statements. The function takes a context, a path p and a set of annotations $\alpha\beta$ and returns a context in which all the super-paths of p have been removed and the annotation of p becomes $\alpha\beta$.

$$\_[\_ \mapsto \_] : \Delta \to p \to \alpha\beta \to \Delta$$

Example 5.5.4.1: Given a context:

$$\Delta = x : \text{unique}, \ x.y : \text{unique}, \ x.y.z : \text{unique}$$

Replace has the following result:

$$\Delta[x.y \mapsto \top] = x : \text{unique}, \ x.y : \top$$

$\square$

5.5.5 Get Super-Paths

$$\text{Get-Super-Paths-Empty} \frac{}{\cdot \vdash \text{superPaths}(p) = \cdot}$$

$$\text{Get-Super-Paths-Discard} \frac{\neg(p \sqsubset p') \qquad \Delta \vdash \text{superPaths}(p) = p_0 : \alpha_0\beta_0, ..., p_n : \alpha_n\beta_n}{p' : \alpha\beta, \Delta \vdash \text{superPaths}(p) = p_0 : \alpha_0\beta_0, ..., p_n : \alpha_n\beta_n}$$

$$\text{Get-Super-Paths-Keep} \frac{p \sqsubset p' \qquad \Delta \vdash \text{superPaths}(p) = p_0 : \alpha_0\beta_0, ..., p_n : \alpha_n\beta_n}{p' : \alpha\beta, \Delta \vdash \text{superPaths}(p) = p' : \alpha\beta, p_0 : \alpha_0\beta_0, ..., p_n : \alpha_n\beta_n}$$

Finally, Get-Super-Paths rules are used to define a function that returns all the super-paths of a give path within a context. Also this function will be used for statements typing rules.

$$\_ \vdash \text{superPaths}(\_) : \Delta \to p \to \overline{p : \alpha\beta}$$

Example 5.5.5.1: Given a context:

$$\Delta = x : \text{unique}, \ x.y : \text{unique}, \ x.y.z : \text{unique}$$

Getting super-paths has the following result:

$$\text{superPaths}(x.y) = x.y.z : \text{unique}$$

$\square$

5.6 Relations between Annotations

5.6.1 Partial Ordering

$$\text{Rel-Id} \frac{}{\alpha\beta \preccurlyeq \alpha\beta} \qquad\qquad \text{Rel-Trans} \frac{\alpha\beta \preccurlyeq \alpha'\beta' \qquad \alpha'\beta' \preccurlyeq \alpha''\beta''}{\alpha\beta \preccurlyeq \alpha''\beta''}$$

$$\text{Rel-Shared-}\flat \frac{}{\text{shared } \flat \preccurlyeq \top} \qquad\qquad \text{Rel-Shared} \frac{}{\text{shared} \preccurlyeq \text{shared } \flat}$$

$$\text{Rel-Unique-}\flat \; \frac{}{\text{unique } \flat \preccurlyeq \text{shared } \flat} \qquad\qquad \text{Rel-Unique-1} \; \frac{}{\text{unique} \preccurlyeq \text{shared}}$$

$$\text{Rel-Unique-2} \; \frac{}{\text{unique} \preccurlyeq \text{unique } \flat}$$

This set of rules is used to define a partial order between the annotations. This partial order can be represented by the lattice shown in Figure 2. The meaning of these relations is that if $\alpha\beta \preccurlyeq \alpha'\beta'$, then $\alpha\beta$ can be used where $\alpha'\beta'$ is expected, for example for method calls. Thanks to these rules, it will be correct to pass a unique reference to a method expecting a shared argument, but not vice versa. Moreover, the relations are consistent with the definition of $\top$ since it will not be possible to pass an inaccessible reference to any method.
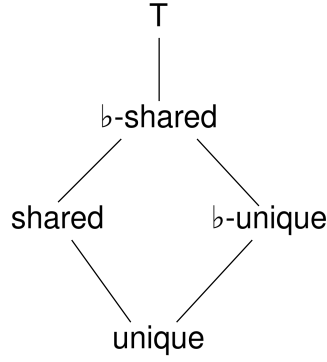
Figure 2: Lattice obtained by Rel rules

5.6.2 Passing

$$\text{Pass-}\flat \; \frac{\alpha\beta \preccurlyeq \alpha'\flat}{\alpha\beta \rightsquigarrow \alpha'\flat \rightsquigarrow \alpha\beta} \qquad\qquad \text{Pass-Unique} \; \frac{}{\text{unique} \rightsquigarrow \text{unique} \rightsquigarrow \top}$$

$$\text{Pass-Shared} \; \frac{\alpha \preccurlyeq \text{shared}}{\alpha \rightsquigarrow \text{shared} \rightsquigarrow \text{shared}}$$

Pass rules define what happens to the annotations of a reference after passing it to a method. If derivable, a judgment $\alpha\beta \rightsquigarrow \alpha'\beta' \rightsquigarrow \alpha''\beta''$ indicates that after passing a reference annotated with $\alpha\beta$ to a method expecting an argument annotated with $\alpha'\beta'$, the reference will be annotated with $\alpha''\beta''$ after the call. However, these rules are not sufficient to type a method call statement since passing the same reference more than once to the same method call is a situation that has to be handled carefully. Nonetheless, the rules are fundamental to express the logic of the annotation system and will be used for typing method calls in subsequent sections.

5.7 Paths

5.7.1 Root

$$\text{Root-Base} \; \frac{}{\operatorname{root}(x) = x} \qquad\qquad \text{Root-Rec} \; \frac{\operatorname{root}(p) = x}{\operatorname{root}(p.f) = x}$$

This simple function takes a path and returns its root. The function can simplify the preconditions of more complex rules. For example $\operatorname{root}(x.y.z) = x$

$$\operatorname{root} : p \to p$$

Example 5.7.1.1:

$$\operatorname{root}(x.y.z) = x$$
$$\operatorname{root}(y.z) = y$$
$$\operatorname{root}(z) = z$$

$\square$

5.7.2 Lookup

$$\text{Lookup-Base} \; \frac{(p : \alpha\beta, \Delta) \text{ ctx}}{(p : \alpha\beta, \Delta)\langle p \rangle = \alpha\beta} \qquad \text{Lookup-Rec} \; \frac{\begin{array}{c} (p : \alpha\beta, \Delta) \text{ ctx} \\ p \neq p' \qquad \Delta\langle p' \rangle = \alpha'\beta' \end{array}}{(p : \alpha\beta, \Delta)\langle p' \rangle = \alpha'\beta'}$$

$$\text{Lookup-Default} \; \frac{\operatorname{default}(f) = \alpha}{\cdot \langle p.f \rangle = \alpha}$$

Lookup rules define a (partial) function that, given a well-formed context, returns the annotations associated with a given path

When the path is explicitly contained within the context, the function returns the corresponding annotation. If a field access $(p.f)$ is not explicitly present in the context, the function returns the annotations specified in the class declaration containing f. This concept, formalized by Lookup-Default, is crucial as it ensures that contexts remain finite, even when handling recursive classes. However, if a variable (x) is not present in the context, its lookup cannot be derived.

It is important to note that the lookup function returns the annotations associated with a path based on the context or the class declaration, rather than determining the actual ownership status of that path.

$$\_\langle \_ \rangle : \Delta \to p \to \alpha\beta$$

Example 5.7.2.1: Given a context:

$$\Delta = x : \text{shared}, \; x.f : \text{unique}$$

The result of the lookup for $x.f$ is the following:

$$\Delta\langle x.f \rangle = \text{unique}$$

However, since x is shared, there can be multiple references accessing x. This implies there can be multiple references accessing $x.f$, meaning that $x.f$ is also shared. A scenario like this can occur when, starting from a context containing only $x : \text{shared}$, a unique value is assigned to the field $x.f$. A function able to determine the actual ownership of a path is defined in the subsequent section.

$\square$

Example 5.7.2.2: Given class C, context Δ and variable x such that:

$$\texttt{class } C(f : \text{shared}) \in P$$
$$\Delta = x : \text{unique}$$

The result of the lookup for $x.f$ is the following:

$$\Delta\langle x.f \rangle = \text{shared}$$

Since $x.f \notin \Delta$, the lookup returns the default annotation, which is the one declared in the class signature. □

5.7.3 Get

$$\text{Get-Var} \frac{\Delta\langle x \rangle = \alpha\beta}{\Delta(x) = \alpha\beta} \qquad\qquad \text{Get-Path} \frac{\Delta(p) = \alpha\beta \qquad \Delta\langle p.f \rangle = \alpha'}{\Delta(p.f) = \bigsqcup\{\alpha\beta, \alpha'\}}$$

As described in the previous subsection, the lookup function might not return the correct annotation for a given path. The task of returning the right annotation for a path within a context is left to the (partial) function described in this section.

$$\_(\_) : \Delta \rightarrow p \rightarrow \alpha\beta$$

In the case that the given path is a variable, the function will return the same annotation returned by the lookup function. If the given path is not a variable, the function will return the least upper bound ($\sqcup$) between the lookup of the given path and all its sub-paths. The LUB between a set of annotations can be easily obtained by using the partial order described in Subsection 5.6.1.

It is important to note that if $\Delta(p) = \alpha\beta$ is derivable for some $\alpha\beta$ then the root of p is contained inside Δ. This is important because many rules in the subsequent sections will use the judgment $\Delta(p) = \alpha\beta$ as a precondition and it also helps to guarantee that the root of p is contained inside Δ.

Furthermore, in the rule Get-Path, the premise $\Delta\langle p.f \rangle = \alpha'$ does not pair a β' annotation to α'. This omission is intentional because, by the design of the subsequent typing rules, a field access lookup should never result in a borrowed annotation. Regardless, the β annotation for a field access should be determined solely by its root.

Example 5.7.3.1: Given a context:

$$\Delta = x : \text{unique}, \ x.y : \top, \ x.y.z : \text{shared}$$

The annotation that is returned for the variable x is the same as the one returned by the lookup.

$$\Delta(x) = \Delta\langle x \rangle = \text{unique}$$

The annotation returned for the path $x.y$ is the LUB between the lookup of $x.y$ and that of all its sub-paths.

$$\Delta(x.y) = \bigsqcup\{\Delta\langle x \rangle, \Delta\langle x.y \rangle\}$$
$$= \bigsqcup\{\text{unique}, \top\}$$
$$= \top$$

Finally, the annotation returned for the path $x.y.z$ is the LUB between the lookup of $x.y.z$ and that of all its sub-paths.

$$\Delta(x.y.z) = \bigsqcup\{\Delta\langle x\rangle, \Delta\langle x.y\rangle, \Delta\langle x.y.z\rangle\}$$
$$= \bigsqcup\{\text{unique}, \top, \text{shared}\}$$
$$= \top$$

$\square$

5.7.4 Standard Form

$$\text{Std-Empty}\ \frac{}{\cdot \vdash \text{std}(p, \alpha\beta)} \qquad\qquad \text{Std-Rec-1}\ \frac{\neg(p \sqsubseteq p') \qquad \Delta \vdash \text{std}(p, \alpha\beta)}{p' : \alpha\beta, \Delta \vdash \text{std}(p, \alpha\beta)}$$

$$\text{Std-Rec-2}\ \frac{\begin{array}{cc} p \sqsubseteq p' & \text{root}(p) = x \\ (x : \alpha\beta)(p') = \alpha''\beta'' & \alpha'\beta' \preccurlyeq \alpha''\beta'' \qquad \Delta \vdash \text{std}(p, \alpha\beta) \end{array}}{p' : \alpha'\beta', \Delta \vdash \text{std}(p, \alpha\beta)}$$

If the judgment $\Delta \vdash \text{std}(p, \alpha\beta)$ is derivable, inside the context Δ, all the super-paths of p carry the right annotations when p is passed to a method expecting an argument annotated with $\alpha\beta$. This type of judgment is necessary verify the correctness of the annotations in a method-modular fashion.

Since a called method does not have information about Δ when verified, all the super-paths of p must have an annotation in Δ that is lower or equal ($\preccurlyeq$) to the annotation that they have in a context containing just their root annotated with $\alpha\beta$.

Example 5.7.4.1: Given the following program:

$$\text{class } C(y : \text{unique})$$
$$m_1(x : \text{unique}) : \text{shared}$$
$$m_2(x : \text{shared}) : \text{shared}$$

Within the context

$$\Delta = x : \text{unique},\ x.y : \text{shared}$$

- $\Delta \vdash \text{std}(x, \text{unique})$ is not derivable, meaning that x cannot be passed to the method m_1. The judgment is not derivable because $\Delta(x.y) = \text{shared}$ while in a context $\Delta' = x : \text{unique}$, $\Delta'(x.y) = \text{unique}$, but shared $\npreccurlyeq$ unique.

- $\Delta \vdash \text{std}(x, \text{shared})$ is derivable, meaning that x can be passed to the method m_2 if all the preconditions, which would be formalized by statement's typing rules, are also satisfied.

$\square$

5.8 Unification

This section introduces several functions essential for managing contexts in control flow constructs such as branching and scope transitions.

5.8.1 Pointwise LUB

$$\text{Ctx-LUB-Empty} \frac{}{\cdot \;\sqcup\; \cdot \;=\; \cdot} \qquad\qquad \text{Ctx-LUB-Sym} \frac{}{\Delta_1 \sqcup \Delta_2 = \Delta_2 \sqcup \Delta_1}$$

$$\text{Ctx-LUB-1} \frac{\Delta_2\langle p\rangle = \alpha''\beta'' \qquad \Delta_2 \setminus p = \Delta_2' \\ \Delta_1 \sqcup \Delta_2' = \Delta' \qquad \bigsqcup\{\alpha\beta, \alpha''\beta''\} = \alpha'\beta'}{(p : \alpha\beta, \Delta_1) \sqcup \Delta_2 = p : \alpha'\beta', \Delta'}$$

$$\text{Ctx-LUB-2} \frac{x \notin \Delta_2 \qquad \Delta_1 \sqcup \Delta_2 = \Delta'}{(x : \alpha\beta, \Delta_1) \sqcup \Delta_2 = x : \top, \Delta'}$$

The rules in this section describe a function that takes two contexts and returns the LUB between each pair of paths in the given contexts. If a variable x is present in only one of the two contexts, it will be annotated with $\top$ in the resulting context.

$$\_ \sqcup \_ : \Delta \to \Delta \to \Delta$$

Example 5.8.1.1:

$$\Delta_1 = x : \text{shared}, \; y : \text{shared}$$
$$\Delta_2 = x : \text{unique}$$
$$\Delta_1 \sqcup \Delta_2 = x : \bigsqcup\{\text{shared}, \text{unique}\}, \; y : \top$$
$$= x : \text{shared}, \; y : \top$$

$\square$

5.8.2 Removal of Local Declarations

$$\text{Remove-Locals-Base} \frac{}{\cdot \blacktriangleleft \Delta = \cdot}$$

$$\text{Remove-Locals-Keep} \frac{\text{root}(p) = x \qquad \Delta_1\langle x\rangle = \alpha'\beta' \qquad \Delta \blacktriangleleft \Delta_1 = \Delta'}{p : \alpha\beta, \Delta \blacktriangleleft \Delta_1 = p : \alpha\beta, \Delta'}$$

$$\text{Remove-Locals-Discard} \frac{\text{root}(p) = x \qquad x \notin \Delta_1 \qquad \Delta \blacktriangleleft \Delta_1 = \Delta'}{p : \alpha\beta, \Delta \blacktriangleleft \Delta_1 = \Delta'}$$

The function formalized by these rules is used to obtain the correct context when exiting a scope. When writing $\Delta_1 \blacktriangleleft \Delta_2$, Δ_1 represents the resulting context of a scope, while Δ_2 represents the context at the beginning of that scope. The result of the operation is a context where paths rooted in variables that have been locally declared inside the scope are removed.

$$\_ \blacktriangleleft \_ : \Delta \to \Delta \to \Delta$$

Example 5.8.2.1:

$$\Delta_1 = x : \text{unique}, \; y : \text{unique}, \; x.f : \text{unique}, \; y.f : \text{shared}$$
$$\Delta_2 = x : \text{shared}$$
$$\Delta_1 \blacktriangleleft \Delta_2 = x : \text{unique}, \; x.f : \text{unique}$$

$\square$

5.8.3 Unify

$$\text{Unify } \frac{\Delta_1 \sqcup \Delta_2 = \Delta_\sqcup \qquad \Delta_\sqcup \blacktriangleleft \Delta = \Delta'}{\text{unify}(\Delta; \Delta_1; \Delta_2) = \Delta'}$$

Finally, the unify function groups the two functions described before. This function will be fundamental to type `if` statements. In particular, $\text{unify}(\Delta; \Delta_1; \Delta_2)$ can be used to type an `if` statement: when Δ is the context at the beginning of the statement while Δ_1 and Δ_2 are the resulting contexts of the two branches of the statement.

$$\text{unify} : \Delta \to \Delta \to \Delta \to \Delta$$

Example 5.8.3.1: Given the following contexts:

$$\Delta = x : \text{unique}$$
$$\Delta_1 = x : \text{shared}, \ x.f : \text{shared}, \ y : \text{unique}$$
$$\Delta_2 = x : \text{unique}, \ x.f : \top, \ y : \text{unique}$$

Unification has the following result:

$$\text{unify}(\Delta; \Delta_1; \Delta_2) = (\Delta_1 \sqcup \Delta_2) \blacktriangleleft \Delta$$
$$= (x : \text{shared}, \ x.f : \top, \ y : \text{unique}) \blacktriangleleft \Delta$$
$$= x : \text{shared}, \ x.f : \top$$

$\square$

5.9 Normalization

$$\text{N-Empty } \frac{}{\text{normalize}(\cdot) = \cdot}$$

$$\text{N-Rec } \frac{\bigsqcup(\alpha_i\beta_i \mid p_i = p_0 \wedge 0 \leq i \leq n) = \alpha_\sqcup\beta_\sqcup \qquad \text{normalize}(p_i : \alpha_i\beta_i \mid p_i \neq p_0 \wedge 0 \leq i \leq n) = p'_0 : \alpha'_0\beta'_0, ..., p'_m : \alpha'_m\beta'_m}{\text{normalize}(p_0 : \alpha_0\beta_0, ..., p_n : \alpha_n\beta_n) = p_0 : \alpha_\sqcup\beta_\sqcup, p'_0 : \alpha'_0\beta'_0, ..., p'_m : \alpha'_m\beta'_m}$$

Normalize is a function that takes and returns a list of annotated paths. In the returned list, duplicate paths from the given list are substituted with a single path annotated with the LUB of the annotations from the duplicate paths. As already mentioned, rules in Subsection 5.6.2 are not sufficient to type a method call because the same path might be passed more than once to the same method. Normalization is the missing piece that will enable the formalization of typing rules for method calls.

$$\text{normalize} : \overline{p : \alpha\beta} \to \overline{p : \alpha\beta}$$

Example 5.9.1:

$$\text{normalize}(x : \top, \ x : \text{shared}, \ y : \text{unique}) = x : \bigsqcup\{\text{shared}, \top\}, \ y : \text{unique}$$
$$= x : \top, \ y : \text{unique}$$

$\square$

5.10 Statements Typing

Typing rules are structured as follows:

$$\Delta \vdash s \dashv \Delta'$$

This judgment means that typing a statement s in a context Δ leads to a context Δ'. It is important to note that this refers only to the types involved and is not related to the operational semantics of the program.

A program P is well-typed if and only if the following judgment is derivable:

$$\forall m\left(\overline{x : \alpha_f \beta}\right) : \alpha_f \{\text{begin}_m; s; \text{return}_m e\} \in P. \quad \cdot \vdash \text{begin}_m; s; \text{return}_m e \dashv \cdot$$

This means that a program is well-typed if and only if, for every method in that program, executing the body of the method within an empty context leads to an empty context. Methods without a body are excluded from this judgment, as they can be safely assumed to be well-typed without further analysis.

5.10.1 Begin

$$\text{Begin} \frac{\text{m-type}(m) = \alpha_0 \beta_0, ..., \alpha_n \beta_n \to \alpha \qquad \text{args}(m) = x_0, ..., x_n}{\cdot \vdash \text{begin}_m \dashv x_0 : \alpha_0 \beta_0, ..., x_n : \alpha_n \beta_n}$$

This rule is used to initialize the context at the beginning of a method. The initial context will contain only the method's parameters with the declared uniqueness annotations. The example below demonstrates how the rule works in practice. In this and subsequent examples, the resulting context after typing a statement is shown on the next line.

```
1  f(this: unique, x: unique ♭, y: shared ♭, z: shared): unique {
2    begin_f;
3      ⊣ Δ = this: unique, x: unique ♭, y: shared ♭, z: shared
4    ...
5  }
```

Listing 31: Typing example for Begin statement

5.10.2 Sequence

$$\text{Seq} \frac{\Delta \vdash s_1 \dashv \Delta_1 \qquad \Delta_1 \vdash s_2 \dashv \Delta'}{\Delta \vdash s_1; s_2 \dashv \Delta'}$$

This rule is straightforward, but necessary to define how to type a sequence of statements. In a sequence, statements are typed in the order that they appear. After a statement is typed, the resulting context is used to type the following one.

5.10.3 Variable Declaration

$$\text{Decl} \frac{x \notin \Delta}{\Delta \vdash \text{var } x \dashv \Delta, x : \top}$$

After declaring a variable, it is inaccessible until its initialization and so the variable will be in the context with $\top$ annotation. Note that this rule only allows to declare variables if they are not in the context while Kotlin allows to shadow variables de-

clared in outer scopes. Kotlin code using shadowing is not currently supported by this system.

```
1  f(): unique {
2    begin_f;
3        ⊣ Δ = ∅
4    var x;
5        ⊣ Δ = x: ⊤
6    ...
7  }
```

Listing 32: Typing example for variable declaration

5.10.4 Call

$$\forall 0 \le i \le n : \Delta(p_i) = \alpha_i \beta_i$$
$$\text{m-type}(m) = \alpha_0^m, \beta_0^m, ..., \alpha_n^m \beta_n^m \to \alpha_r$$
$$\forall 0 \le i \le n : \Delta \vdash \text{std}(p_i, \alpha_i^m \beta_i^m)$$
$$\forall 0 \le i, j \le n : \left(i \neq j \wedge p_i = p_j \right) \Rightarrow \alpha_i^m = \text{shared}$$
$$\forall 0 \le i, j \le n : p_i \sqsubset p_j \Rightarrow \left(\Delta(p_j) = \text{shared} \vee \alpha_i^m = \alpha_j^m = \text{shared} \right)$$
$$\Delta \ominus (p_0, ..., p_n) = \Delta' \qquad \forall 0 \le i \le n : \alpha_i \beta_i \rightsquigarrow \alpha_i^m \beta_i^m \rightsquigarrow \alpha_i' \beta_i'$$
$$\text{Call} \frac{\text{normalize}(p_0 : \alpha_0' \beta_0', ..., p_n : \alpha_n' \beta_n') = p_0' : \alpha_0'' \beta_0'', ..., p_m' : \alpha_m'' \beta_m''}{\Delta \vdash m(p_0, ..., p_n) \dashv \Delta', p_0' : \alpha_0'' \beta_0'', ..., p_m' : \alpha_m'' \beta_m''}$$

Typing a method call follows the logic presented in the rules of Subsection 5.6.2 ($\rightsquigarrow$) while taking care of what can happen with method accepting multiple parameters.

- All the roots of the paths passed to a method must be in the context (also guaranteed by the language).
- All the paths passed to a method must be in standard form of the expected annotation.
- It is allowed to pass the same path twice to the same method, but only if it passed where a shared argument is expected.
- It is allowed to pass two paths p_i and p_j such that $p_i \sqsubset p_j$ when one of the following conditions is satisfied:
  ‣ p_j is shared.
  ‣ The method that has been called expects shared (possibly borrowed) arguments in positions i and j.
- The resulting context is constructed in the following way:
  ‣ Paths passed to the method and their super-paths are removed from the initial context.
  ‣ A list of annotated paths (in which a the same path may appear twice) in constructed by mapping passed paths according to the "passing" ($\rightsquigarrow$) rules.
  ‣ The obtained list is normalized and added to the context.

Listing 33 shows the cases where it is possible to pass the same reference more than once and how normalization is applied. In Listing 34 it is possible to call f by passing x and x.f since $\Delta(x.f) = \text{shared}$. In Listing 35 is not possible to call g by passing b and b.f, this is because g, in its body, expects x.f to be unique, but it would not be the case by passing b and b.f. Finally Listing 36 shows that it is possible to call h by passing x and x.f since the method expects both of the arguments to be shared.

46

```
1   f(x: unique, y: shared ♭): unique
2
3   g(x: shared ♭, y: shared ♭): unique
4
5   h(x: shared, y: shared ♭): unique
6
7   use_f(x: unique) {
8     begin_use_f;
9       ⊣ Δ = x: unique
10    f(x, x);
11    // not derivable: 'x' is passed more than once but is also expected to
      be unique
12    ...
13  }
14
15  use_g_h(x: unique) {
16    begin_use_g_h;
17      ⊣ Δ = x: unique
18    g(x, x); // ok, uniqueness is also preserved since both the args are
      borrowed
19      ⊣ Δ = x: unique
20    h(x, x); // ok, but uniqueness is lost after normalization
21      ⊣ Δ = x: shared
22  }
```

Listing 33: Typing example for method call with same reference

```
1   class A(f: shared)
2
3   f(x: unique, y: shared): unique
4
5   fun use_f(x: unique) {
6     begin_use_f;
7       ⊣ Δ = x: unique
8     f(x, x.f); // ok
9       ⊣ Δ = x: T, x.f: shared
10    // Note that even if x.f is marked shared in the context, it is not
      accessible since Δ(x.f) = T
11    ...
12  }
```

Listing 34: Typing example for correct method call with super-paths

```
1   class B(f: unique)
2
3   g(x: unique, y: shared): unique
4
5   use_g(b: unique) {
6     begin_use_g;
7       ⊣ Δ = b: unique
8     g(b, b.f);
9      // error: 'b.f' cannot be passed since 'b' is passed as unique and
    Δ(b.f) = unique
10    // It is correct to raise an error since 'g' expects x.f to be unique
11  }
```

Listing 35: Typing example for incorrect method call with super-paths

```
1   class B(f: unique)
2
3   h(x: shared, y: shared) {}
4
5   use_h(x: unique) {
6     begin_use_h;
7       ⊣ Δ = x: unique
8     h(x, x.f); // ok
9       ⊣ Δ = x: shared, x.f: shared
10    ...
11  }
```

Listing 36: Typing example for correct method call with super-paths

5.10.5 Assignments

All rules for typing assignments have a path p on the left-hand side and vary based on the expression on the right-hand side. The common trait of these rules is that they require the root of p to be contained within the initial context using the premise "$\Delta(p) = \alpha\beta$". Additionally, in the resulting context, the annotation of p is always updated according to the expression on the right-hand side of the assignment.

5.10.5.1 Assign null

$$\text{Assign-Null} \frac{\Delta(p) = \alpha\beta \qquad \Delta[p \mapsto \text{unique}] = \Delta'}{\Delta \vdash p = \text{null} \dashv \Delta'}$$

The definition of unique tells us that a reference is unique when it is null or is the sole accessible reference pointing to the object that is pointing. Given that, we can safely consider unique a path p after assigning null to it. Moreover, all super-paths of p are removed from the context after the assignment.

```
1   class C(t: unique)
2
3   f() {
4     begin_f;
5        ⊣ Δ = ∅
6     var b;
7        ⊣ Δ = b: T
8     ...
9        ⊣ Δ = b: shared, b.t: T
10    b = null
11       ⊣ Δ = b: unique
12    ...
13  }
```

Listing 37: Typing example for assigning null

5.10.5.2 Assign Call

$$
\text{Assign-Call} \frac{
\begin{array}{cc}
\Delta(p) = \alpha'\beta' & \Delta \vdash m(\overline{p}) \dashv \Delta_1 \\
\multicolumn{2}{c}{\text{m-type}(m) = \alpha_0\beta_0, ..., \alpha_n\beta_n \to \alpha} \\
(\beta' = \flat) \Rightarrow (\alpha = \text{unique}) & \Delta_1[p \mapsto \alpha] = \Delta'
\end{array}
}{
\Delta \vdash p = m(\overline{p}) \dashv \Delta'
}
$$

After defining how to type a method call, it is easy to formalize the typing of a call assignment. Like all the other assignment rules, the root of the path on the left side of the assignment must be in the context. First of all, the method call is typed obtaining a new context Δ_1. Then, the annotation of the path on the left side of the assignment is replaced ($\mapsto$) in Δ_1 with the annotation of the return value of the method.

```
1   get_unique(): unique
2   get_shared(): shared
3
4   f(): unique {
5     begin_f;
6        ⊣ Δ = ∅
7     var x;
8        ⊣ Δ = x: T
9     var y;
10       ⊣ Δ = x: T, y: T
11    x = get_unique();
12       ⊣ Δ = x: unique, y: T
13    y = get_shared();
14       ⊣ Δ = x: unique, y: shared
15    ...
16  }
```

Listing 38: Typing example for assigning a method call

5.10.5.3 Assign Unique

$$\text{Assign-Unique} \frac{p' \not\sqsubseteq p \qquad \Delta(p) = \alpha\beta \qquad \Delta(p') = \text{unique} \qquad \Delta[p' \mapsto \top] = \Delta_1}{\Delta \vdash \text{superPaths}(p') = p'.\overline{f_0} : \alpha_0\beta_0, ..., p'.\overline{f_n} : \alpha_n\beta_n \qquad \Delta_1[p \mapsto \text{unique}] = \Delta'}{\Delta \vdash p = p' \dashv \Delta', p.\overline{f_0} : \alpha_0\beta_0, ..., p.\overline{f_n} : \alpha_n\beta_n}$$

In order to type an assignment $p = p'$ in which p' is unique, the following conditions must hold:
- The root of p must be in context.
- p' must be unique in the context.
- Assignments in which $p' \sqsubseteq p$, like $p.f = p$, are not allowed.

The resulting context is built in the following way:
- Starting from the initial context Δ, a context Δ_1 is obtained by replacing ($\mapsto$) the annotation of p' with $\top$.
- The context Δ_1 is used to obtain a context Δ' by replacing ($\mapsto$) the annotation of p with unique.
- Finally, to obtain the resulting context, all the paths that were originally rooted in p' are rooted in p with the same annotation and added to Δ'.

```
1   class B(t: unique)
2   class A(b: unique)
3
4   f(x: unique, y: unique): unique {
5     begin_f;
6       ⊣ Δ = x: unique, y: unique
7     y.t = x.b.t;
8       ⊣ Δ = x: unique, y: unique, x.b.t: T, y.t: unique
9     x.b = y;
10      ⊣ Δ = x: unique, y: T, x.b: unique
11    ...
12  }
```

Listing 39: Typing example for assigning a unique reference

5.10.5.4 Assign Shared

$$\text{Assign-Shared} \frac{p' \not\sqsubseteq p \qquad \Delta(p) = \alpha \qquad \Delta(p') = \text{shared}}{\Delta \vdash \text{superPaths}(p') = p'.\overline{f_0} : \alpha_0\beta_0, ..., p'.\overline{f_n} : \alpha_n\beta_n \qquad \Delta[p \mapsto \text{shared}] = \Delta'}{\Delta \vdash p = p' \dashv \Delta', p.\overline{f_0} : \alpha_0\beta_0, ..., p.\overline{f_n} : \alpha_n\beta_n}$$

Typing an assignment $p = p'$ in which p' is shared is similar to the case where p' is unique, but with some differences:
- p cannot be borrowed. This is necessary to guarantee the soundness of the system when a unique variable is passed to a method expecting a shared borrowed argument.
- Obviously p' must be shared in the context.

Also the resulting context is constructed in a similar way to the previous case. The only difference is that in this case it is not needed to replace ($\mapsto$) the annotation of p'.

```
1  class B(t: unique)
2
3  f(x: unique, y: shared): unique {
4    begin_f;
5      ⊣ Δ = x: unique, y: shared
6    x.t = y;
7      ⊣ Δ = x: unique, y: shared, x.t: shared
8    ...
9  }
```

Listing 40: Typing example for assigning a shared reference

5.10.5.5 Assign Borrowed Field

$$p'.f \not\sqsubseteq p \qquad \Delta(p) = \alpha\beta \qquad \Delta(p'.f) = \alpha'\flat$$

$$\alpha' \neq \top \qquad (\beta = \flat) \Rightarrow (\alpha' = \text{unique}) \qquad \Delta[p'.f \mapsto \top] = \Delta_1$$

$$\text{Assign-}\flat\text{-Field} \frac{\Delta \vdash \text{superPaths}(p'.f) = p'.f.\overline{f_0} : \alpha_0\beta_0, ..., p'.f.\overline{f_n} : \alpha_n\beta_n \qquad \Delta_1[p \mapsto \alpha'] = \Delta'}{\Delta \vdash p = p'.f \dashv \Delta', p.\overline{f_0} : \alpha_0\beta_0, ..., p.\overline{f_n} : \alpha_n\beta_n}$$

Fields of a borrowed parameter must be treated with caution to avoid unsoundness. Borrowed fields can be passed as arguments to other methods if the preconditions for typing the method call are respected. In addition, they can be used on the right-hand side of an assignment with certain limitations. After being read, a borrowed field will inaccessible even if shared. Finally, borrowed fields can be used on the left-hand side of an assignment when a unique reference is on the right-hand side.

Ensuring inaccessibility after reading borrowed fields and restricting their reassignment to unique references, along with respecting the preconditions for typing a return statement stated in Subsection 5.10.7, is essential for maintaining soundness when unique references are passed to methods that accept a borrowed-shared parameter.

```
1   class B(t: unique)
2
3   f(x: shared ♭): unique {
4     begin_f;
5       ⊣ Δ = x: shared ♭,
6     var z;
7       ⊣ Δ = x: shared ♭, z: ⊤
8     z = x.t;
9       ⊣ Δ = x: shared ♭, z: shared, x.t: ⊤
10    ...
11  }
```

Listing 41: Typing example for assigning a borrowed field

5.10.6 If

$$\Delta(p_1) \neq \top \qquad \Delta(p_2) \neq \top$$

$$\text{If} \frac{\Delta \vdash s_1 \dashv \Delta_1 \qquad \Delta \vdash s_2 \dashv \Delta_2 \qquad \text{unify}(\Delta; \Delta_1; \Delta_2) = \Delta'}{\Delta \vdash \text{if } p_1 == p_2 \text{ then } s_1 \text{ else } s_2 \dashv \Delta'}$$

Once the unification function is defined, typing an `if` statement is straightforward. First it is necessary to be sure that paths appearing in the guard are accessible in the initial context. The `then` and the `else` branches are typed separately and their resulting contexts are unified to get the resulting context of the whole statement. The system does not allow to have `null` or a method call in the guard of an `if` statement, as these constructs can be easily desugared.

> **Example 5.10.6.1**: Desugaring for `if` statements containing expressions different from paths within the guard.
>
> $$\text{if } (p == \text{null})... \equiv \text{var fresh}; \text{fresh} = \text{null}; \text{if}(p == \text{fresh})...$$
>
> $$\text{if } (p == m(...))... \equiv \text{var fresh}; \text{fresh} = m(...); \text{if}(p == \text{fresh})...$$
>
> □

```
1   class A(c: unique)
2
3   consume_unique(c: unique): shared
4
5   consume_shared(a: shared): shared
6
7   fun f(a: unique, c: shared ♭) {
8     begin_f;
9        ⊣ Δ = a: unique, t: shared ♭
10    if (a.c == c) {
11        consume_unique(a.c);
12           ⊣ Δ1 = a: unique, a.f: T, t: shared ♭
13    } else {
14        consume_shared(a);
15           ⊣ Δ2 = a: shared, t: shared ♭
16    };
17       ⊣ Δ = a: shared, a.f: T, t: shared ♭
18    // unify(Δ; Δ1; Δ2) = a: LUB{ unique, shared }, a.f: LUB{ T, shared },
       t: shared ♭
19    ...
20  }
```

Listing 42: Typing example for if statement

5.10.7 Return

$$\text{Return-p} \frac{\begin{array}{ccc} \text{m-type}(m) = \alpha_0^m, \beta_0^m, ..., \alpha_n^m \beta_n^m \to \alpha_r & \Delta(p) = \alpha\beta & \alpha\beta \preccurlyeq \alpha_r \\ \Delta \vdash \text{std}(p, \alpha_r) & \forall 0 \le i, j \le n : (\alpha_i \beta_i \ne \text{unique}) \Rightarrow \Delta \vdash \text{std}(p_i, \alpha_i \beta_i) \end{array}}{\Delta \vdash \text{return}_m \ p \dashv \cdot}$$

By the construction of the grammar, a `return` statement is designed to be the final statement executed within a method. As such, there is no need to maintain a resulting context after the return statement has been typed. However, several important conditions must be satisfied when returning.

First, the annotation of the path being returned must be lower than or equal to ($\preccurlyeq$) the annotation of the return value of the method. This ensures that a method cannot return a value with greater aliasing than what was specified in the method's signature, effectively preventing borrowed values from being returned (Example 5.10.7.1).

Second, the path being returned must be in the standard form of the return type (Example 5.10.7.2).

Finally, all parameters that are shared or borrowed (or both) must remain in the standard form of their original annotations by the time the method returns.

These conditions are essential for maintaining the modularity, allowing each method to be typed without knowing the implementation of the other methods.

The system does not allow returning null or a method call, since these cases can be easily desugared, as shown in Example 5.10.7.3. Similarly, functions that do not return a value can be represented by having them return a call to the Unit constructor.

Example 5.10.7.1: Given the following program:

$$\text{class } C(f : \text{unique})$$
$$m(x : \text{unique } \flat) : \text{unique } \{\text{begin}_m; ...; \text{return}_m \ x.f\}$$

The following judgment is not derivable:

$$x : \text{unique } \flat = \Delta \vdash \text{return}_m \ x.f \dashv \cdot$$

This happens because the function returns a borrowed field, which is prohibited by the third precondition of the rule. Specifically:

$$\text{m-type}(m) = \text{unique } \flat \rightarrow \text{unique}$$
$$\Delta(x.f) = \text{unique } \flat$$

However, the third precondition is not derivable since:

$$\text{unique } \flat \npreccurlyeq \text{unique}$$

$\square$

Example 5.10.7.2: Given the following program:

$$\text{class } C(f : \text{unique})$$
$$m(x : \text{unique}) : \text{unique } \{\text{begin}_m; ...; \text{return}_m \ x\}$$

The following judgment is not derivable:

$$x : \text{unique}, \ x.f : \text{shared} = \Delta \vdash \text{return}_m \ x \dashv \cdot$$

This occurs because the fourth precondition, $\Delta \vdash \text{std}(x, \text{unique})$, is not derivable.

$\square$

Example 5.10.7.3: Desugaring for return statements that do not return a path.

$$\{...; \text{return null}\} \equiv \{...; \text{var fresh}; \text{fresh} = \text{null}; \text{return fresh}\}$$

$$\{...; \text{return } m(...)\} \equiv \{...; \text{var fresh}; \text{fresh} = m(...); \text{return fresh}\}$$

Where fresh refers to a variable that does not exist in the context prior to its declaration. $\square$

5.11 Stack Example

Listing 43 illustrates how the context evolves in the example presented in Section 4.3 when it is encoded using the grammar described in this chapter.

```
1   class Node(value: unique, next: unique)
2
3   class Stack(root: unique)
4
5   fun Node(value: unique, next: unique): unique
6
7   fun push(this: unique ♭, value: unique): shared {
8     begin_push;
9       ⊣ Δ = this: unique ♭, value: unique
10    var r;
11      ⊣ Δ = this: unique ♭, value: unique, r: T
12    r = this.root;
13      ⊣ Δ = this: unique ♭, value: unique, r: unique, this.root: T
14    this.root = Node(value, r);
15      ⊣ Δ = this: unique ♭, value: T, r: T, this.root: unique
16    return Unit();
17  }
18
19  fun pop(this: unique ♭): unique {
20    begin_pop;
21      ⊣ Δ = this: unique ♭
22    var value;
23      ⊣ Δ = this: unique ♭, value: T
24    if (this.root == null) {
25        value = null;
26          ⊣ Δ = this: unique ♭, value: unique
27    } else {
28        value = this.root.value;
29          ⊣ Δ = this: unique ♭, value: unique, this.root.value: T
30        this.root = this.root.next;
31          ⊣ Δ = this: unique ♭, value: unique, this.root: unique
32    }
33    // Unification...
34      ⊣ Δ = this: unique ♭, value: unique, this.root: unique
35    return value;
36  }
```

Listing 43: Typing for a Stack implementation

Chapter 6

Encoding in Viper

The annotation system for aliasing control introduced in Chapter 4 and formalized in Chapter 5 aims to improve the verification process performed by SnaKt [20], an existing plugin for the Kotlin compiler. SnaKt verifies Kotlin code by encoding it to Viper and supports a substantial subset of the Kotlin language. However, as described in Section 2.2, the lack of guarantees about aliasing presents a significant limitation for the plugin. This chapter illustrates how uniqueness annotations can be used to improve the encoding of Kotlin into Viper.

6.1 Classes Encoding

In Kotlin, as in most programming languages, classes can represent potentially unbounded structures on the heap, such as linked lists or trees. This characteristic was a key factor in the decision to encode Kotlin classes into Viper predicates. Viper predicates, in fact, are specifically designed to represent potentially unbounded data structures.

6.1.1 Shared Predicate

The shared predicate of a class includes read access to all fields that the language guarantees as immutable. Having access to these predicates allows the verification of certain functional properties of a program, even without uniqueness guarantees. The reason is that immutability is a stronger condition than uniqueness from a verification point of view. Indeed, while uniqueness ensures that an object is only accessible through a single reference, immutability guarantees that the object's state cannot change after its creation, eliminating the need to track or control access patterns for verifying correctness.

As shown in Listing 44, the encoding process involves including access to all fields declared as `val`, along with their shared predicate if they have one. Inheritance is encoded by including access to the shared predicates of the supertypes (Line 18). Additionally, the example illustrates how Kotlin's nullable types are encoded by accessing the predicate when the reference is not `null` through a logical implication (Lines 19-21).

All the encoding examples that follow are simplified to improve readability and focus on the aspects pertinent to this work. In the plugin, avoiding name clashes is a crucial concern. As a result, names generated by SnaKt are typically more complex than those shown in the examples.

Furthermore, SnaKt extends the predicate's body by incorporating Kotlin type information using domain functions. Unlike other assertions within the predicate, these domain functions are not resource assertions but rather logical assertions about the Kotlin type of a reference. For example, instead of directly mapping Kotlin `Int` and other primitive types to their corresponding built-in Viper types, SnaKt maps them to Viper `Ref` type, each associated with a domain function that asserts the specific Kotlin type of the reference. This approach ensures that the type information is logically represented within the verification process. Similarly, when dealing with classes, SnaKt

pairs them with domain functions to maintain consistent type information throughout the verification. This representation, while crucial for accurate type tracking, is omitted in the examples provided here, as it is not central to the primary focus of this work. For a complete view, Listing 45 shows how the shared predicate of the classes in Listing 44 appears in the plugin.

```kotlin
1   open class A(
2       val x: Int,
3       var y: Int,
4   )
5
6   class B(
7       val a1: A,
8       var a2: A,
9   )
10
11  class C(
12      val b: B?,
13  ) : A(0, 0)
```

```viper
1   field x: Int
2   field y: Int
3   field a1: Ref
4   field a2: Ref
5   field b: Ref
6
7   predicate SharedA(this: Ref) {
8     acc(this.x, wildcard)
9   }
10
11  predicate SharedB(this: Ref) {
12    acc(this.a1, wildcard) &&
13    acc(SharedA(this.a1), wildcard)
14  }
15
16  predicate SharedC(this: Ref) {
17    acc(SharedA(this), wildcard)
18    acc(this.b, wildcard) &&
19    (this.b != null ==>
20    acc(SharedB(this.b), wildcard))
21  }
```

Listing 44: Shared predicate encoding

```viper
1   field bf$a1: Ref                                          🐦 Viper
2   field bf$a2: Ref
3   field bf$b: Ref
4   field bf$x: Ref
5   field bf$y: Ref
6
7   predicate p$c$A$shared(this: Ref) {
8     acc(this.bf$x, wildcard) &&
9     df$rt$isSubtype(df$rt$typeOf(this.bf$x), df$rt$intType())
10  }
11
12  predicate p$c$B$shared(this: Ref) {
13    acc(this.bf$a1, wildcard) &&
14    acc(p$c$A$shared(this.bf$a1), wildcard) &&
15    df$rt$isSubtype(df$rt$typeOf(this.bf$a1), df$rt$T$c$A())
16  }
17
18  predicate p$c$C$shared(this: Ref) {
19    acc(this.bf$b, wildcard) &&
20    (this.bf$b != df$rt$nullValue() ==>
21    acc(p$c$B$shared(this.bf$b), wildcard)) &&
22    acc(p$c$A$shared(this), wildcard) &&
23   df$rt$isSubtype(df$rt$typeOf(this.bf$b), df$rt$nullable(df$rt$T$c$B()))
24  }
```

Listing 45: Shared predicate non-simplified encoding

6.1.2 Unique Predicate

The unique predicate of a class grants access to all its fields with either `write` or `wildcard` permission, depending on whether the field is declared as `var` or `val`. If a field is marked as unique, the unique predicate also includes access to that field's unique predicate. Additionally, the predicate contains access assertions to the shared predicates of the fields since, as explained in the previous section, accessing immutable resources is always safe.

It is worth mentioning that some overlap might exist between the assertions in the shared predicate and those in the unique predicate. However, this overlap cannot lead to contradictions in Viper, such as requiring access with a total amount greater than 1, because the only assertions that can overlap are accessed with `wildcard` permission.

```kotlin
1   class A(                    Kotlin
2       val x: Int,
3       var y: Int,
4   )
5
6   class B(
7       @property:Unique
8       val a1: A,
9       val a2: A,
10  )
11
12
13
14  class C(
15      @property:Unique
16      val b: B?,
17  ) : A(0, 0)
```

```
1   predicate UniqueA(this: Ref) {   Viper
2       acc(this.x, wildcard) &&
3       acc(this.y, write)
4   }
5
6   predicate UniqueB(this: Ref) {
7       acc(this.a1, wildcard) &&
8       acc(SharedA(this.a1), wildcard) &&
9       acc(UniqueA(this.a1), write) &&
10      acc(this.a2, wildcard) &&
11      acc(SharedA(this.a2), wildcard) &&
12  }
13
14  predicate UniqueC(this: Ref) {
15      acc(this.b, wildcard) &&
16      (this.b != null ==>
17      acc(SharedB(this.b), wildcard)) &&
18      (this.b != null ==>
19      acc(UniqueB(this.b), write)) &&
20      acc(UniqueA(this), write)
21  }
```

Listing 46: Unique predicate encoding

6.2 Functions Encoding

Access information provided by Kotlin's type system and by the uniqueness annotations is encoded using the predicates described in Section 6.1 within the conditions of a method. On the one hand, shared predicates can always be accessed with `wildcard` permission without causing issues. Therefore, they can always be included in the conditions of a method for its parameters, receiver, and return value. On the other hand, unique predicates can only be included in a method's conditions in accordance with the annotation system.

6.2.1 Return object

Since accessing immutable data is not a problem even if it is shared, every Kotlin function, in its encoding can ensure access to the shared predicate of the type of the returned object. In addition, a Kotlin function annotated to return a unique object will also ensure access to its unique predicate. Listing 47 illustrates the differences in the encoding between a function that returns a unique object and a function that returns a shared one.

```
1  @Unique                    🄺 Kotlin         1  method returnUnique()              🐍 Viper
2  fun returnUnique(): T {                       2  returns(ret: Ref)
3      // ...                                    3  ensures acc(SharedT(ret), wildcard)
4  }                                             4  ensures acc(UniqueT(ret), write)
5                                                5
6  fun returnShared(): T {                       6  method returnShared()
7      // ...                                    7  returns(ret: Ref)
8  }                                             8  ensures acc(SharedT(ret), wildcard)
```

Listing 47: Function return object encoding

6.2.2 Parameters

Annotations on parameters are encoded by adding preconditions and postconditions to the method. Access to the shared predicate of any parameter can always be required in preconditions and ensured in postconditions. Conversely, access to the unique predicate can be required in preconditions only for parameters annotated as unique, and it can be ensured in postconditions only for parameters annotated as both unique and borrowed. Listing 48 shows how function parameters are encoded, while Table 1 summarizes the assertions contained within preconditions and postconditions based on the parameter annotations.

In Kotlin, when passing a unique reference to a function that expects a shared borrowed argument, fields included in the unique predicate can still be modified. The current encoding does not fully capture this behavior. However, as shown in Section 6.4, this limitation can be addressed by adding additional statements when such functions are called.

```
1   fun arg_unique(        🄺 Kotlin       1   method arg_unique(t: Ref)      🐍 Viper
2       @Unique t: T                       2   requires acc(UniqueT(t))
3   ) {                                    3   requires acc(SharedT(t), wildcard)
4   }                                      4   ensures acc(SharedT(t), wildcard)
5                                          5
6   fun arg_shared(                        6   method arg_shared(t: Ref)
7       t: T                               7   requires acc(SharedT(t), wildcard)
8   ) {                                    8   ensures acc(SharedT(t), wildcard)
9   }                                      9
10                                         10  method arg_unique_b(t: Ref)
11  fun arg_unique_b(                      11  requires acc(UniqueT(t))
12      @Unique @Borrowed t: T             12  requires acc(SharedT(t), wildcard)
13  ) {                                    13  ensures acc(UniqueT(t))
14  }                                      14  ensures acc(SharedT(t), wildcard)
15                                         15
16  fun arg_shared_b(                      16  method arg_shared_b(t: Ref)
17      @Borrowed t: T                     17  requires acc(SharedT(t), wildcard)
18  ) {                                    18  ensures acc(SharedT(t), wildcard)
19  }
```

Listing 48: Function parameters encoding

	Unique	Unique Borrowed	Shared	Shared Borrowed
Requires Shared Predicate	✓	✓	✓	✓
Ensures Shared Predicate	✓	✓	✓	✓
Requires Unique Predicate	✓	✓	✗	✗
Ensures Unique Predicate	✗	✓	✗	✗

Table 1: Conditions for annotated parameters

6.2.3 Receiver

Encoding the receiver of a method is straightforward since the receiver is considered as a normal parameter.

```kotlin
1  fun @receiver:Unique T.uniqueReceiver() {}
2
3  fun @receiver:Unique @receiver:Borrowed T.uniqueBorrowedReceiver() {}
```

```viper
1   method uniqueReceiver(this: Ref)
2   requires acc(SharedT(this), wildcard)
3   requires acc(UniqueT(this), write)
4   ensures acc(SharedT(this), wildcard)
5
6   method uniqueBorrowedReceiver(this: Ref)
7   requires acc(SharedT(this), wildcard)
8   requires acc(UniqueT(this), write)
9   ensures acc(SharedT(this), wildcard)
10  ensures acc(UniqueT(this), write)
```

Listing 49: Function receiver encoding

6.2.4 Constructor

Constructors are encoded as black-box methods returning a unique object. The encoding of a constructor requires access to the shared predicates for every property that is not of a primitive type. In addition, the unique predicate is also required for properties that are unique in the class declaration. Currently, SnaKt only supports class properties declared as parameters. Properties declared within the body of a class and initializing blocks are not supported yet, as they may construct objects that are not necessarily unique.

60

```kotlin
1  class A(val x: Int, var y: Int)                                ⓚ Kotlin
2
3  class B(@property:Unique var a1: A, var a2: A)
```

```viper
1   method constructorA(p1: Int, p2: Int) returns (ret: Ref)      ⓥ Viper
2     ensures acc(SharedA(ret), wildcard)
3     ensures acc(UniqueA(ret), write)
4     ensures unfolding acc(SharedA(ret), wildcard) in
5       ret.x == p1
6     ensures unfolding acc(UniqueA(ret), write) in
7       ret.x == p1 && ret.y == p2
8
9   method constructorB(p1: Ref, p2: Ref) returns (ret: Ref)
10    requires acc(UniqueA(p1), write)
11    requires acc(SharedA(p1), wildcard)
12    requires acc(SharedA(p2), wildcard)
13    ensures acc(SharedB(ret), wildcard)
14    ensures acc(UniqueB(ret), write)
15    ensures unfolding acc(UniqueB(ret), write) in
16      ret.a1 == p1 && ret.a2 == p2
```

Listing 50: Constructor encoding

6.3 Accessing Properties

While encoding the body of a function using predicates to represent classes, multiple `unfold`, `fold`, `inhale`, and `exhale` statements may be necessary to access the properties of a class. If a property is part of a shared predicate, it is accessed through that predicate. If no shared predicate contains the property, the plugin attempts to access it through a unique predicate, if available. If the property is not even contained within any unique predicate, the access is inhaled.

6.3.1 Accessing Properties within Shared Predicate

Accessing properties contained within a shared predicate is straightforward. This is because shared predicates are always accessed with `wildcard` permission, meaning that after unfolding, the predicate remains valid, so there is no need to fold it back. In Listing 51, it is possible to note that the encoding of the function `f` does not require folding any predicate after accessing `b.a.n` to satisfy its postconditions.

```kotlin
1   class A(                    🅺 Kotlin
2       val n: Int
3   )
4
5   class B(
6       val a: A
7   )
8
9   fun f(b: B): Int {
10      return b.a.n
11  }
```

```viper
1   field n: Int, a: Ref           ⓥ Viper
2
3   predicate SharedA(this: Ref) {
4     acc(this.n, wildcard)
5   }
6
7   predicate SharedB(this: Ref) {
8     acc(this.a, wildcard) &&
9     acc(SharedA(this.a), wildcard)
10  }
11
12  method f(b: Ref) returns(res: Int)
13  requires acc(SharedB(b), wildcard)
14  ensures acc(SharedB(b), wildcard)
15  {
16    unfold acc(SharedB(b), wildcard)
17    unfold acc(SharedA(b.a), wildcard)
18    res := b.a.n
19  }
```

Listing 51: Immutable property access encoding

6.3.2 Accessing Properties within Unique Predicate

When accessing a property through a unique predicate, the predicate must be unfolded with `write` permission. Unlike shared predicates, which remain valid after unfolding with `wildcard` permission, a unique predicate does not hold after it has been unfolded. If the unique predicate is needed again, it must be folded back. This is necessary when satisfying the postconditions of the method or the preconditions of a called method.

```kotlin
                                    🄺 Kotlin
1  class A(
2      var n: Int
3  )
4
5  class B(
6      @property:Unique
7      var a: A
8  )
9
10 fun f(
11     @Unique b: B
12 ): Int {
13     return b.a.n
14 }
```

```viper
                                     🅥 Viper
1  field n: Int, a: Ref
2
3  predicate UniqueA(this: Ref) {
4    acc(this.n, write)
5  }
6
7  predicate UniqueB(this: Ref) {
8    acc(this.a, write) &&
9    acc(UniqueA(this.a), write)
10 }
11
12 method f(b: Ref) returns(res: Int)
13 requires acc(UniqueB(b), write)
14 ensures acc(UniqueB(b), write)
15 {
16   unfold acc(UniqueB(b), write)
17   unfold acc(UniqueA(b.a), write)
18   res := b.a.n
19   fold acc(UniqueA(b.a), write)
20   fold acc(UniqueB(b), write)
21 }
```

Listing 52: Unique mutable property access encoding

6.3.3 Accessing Properties not Contained within a Predicate

When no predicates contain the access to a property that needs to be accessed, it must be inhaled. After the property is used, its access is immediately exhaled. It is important to note that once the access to a property is exhaled, all information about it is lost. This is coherent with the idea that a property not contained within a predicate is mutable and shared, making it impossible to reason about it. In fact, such a property could be accessed and modified by other functions running concurrently.

```kotlin
1   class A(                    🅚 Kotlin
2       val x: Int,
3       var y: Int
4   )
5
6   fun f(
7       a: A
8   ) {
9       a.y = 1
10  }
```

```viper
1   field x: Int, y: Int              🅥 Viper
2
3   predicate SharedA(this: Ref) {
4     acc(this.x, wildcard)
5   }
6
7   method f(a: Ref) returns(res: Int)
8   requires acc(SharedA(a), wildcard)
9   ensures acc(SharedA(a), wildcard)
10  {
11    inhale acc(a.y, write)
12    a.y := 1
13    exhale acc(a.y, write)
14  }
```

Listing 53: Shared mutable property access encoding

6.4 Function Calls Encoding

Encoding method calls is straightforward for some cases, but requires attention for some others.

6.4.1 Functions with Unique Parameters

Functions with a unique parameter, when called, do not need the inclusion of additional statements for their encoding, except for folding or unfolding statements, as detailed in Section 6.3.

```kotlin
1   fun uniqueParam(           🅚 Kotlin
2       @Unique t: T
3   ) {
4   }
5
6   fun uniqueBorrowedParam(
7       @Unique @Borrowed t: T
8   ) {
9   }
10
11  fun call(
12      @Unique @Borrowed t1: T,
13      @Unique t2: T
14  ) {
15      uniqueBorrowedParam(t1)
16      uniqueBorrowedParam(t2)
17      uniqueParam(t2)
18  }
```

```viper
1   method uniqueParam(t: Ref)    🅥 Viper
2   requires acc(UniqueT(t), write) &&
    acc(SharedT(t), wildcard)
3   ensures acc(SharedT(t), wildcard)
4
5   method uniqueBorrowedParam(t: Ref)
6   requires acc(UniqueT(t), write) &&
    acc(SharedT(t), wildcard)
7   ensures acc(UniqueT(t), write) &&
    acc(SharedT(t), wildcard)
8
9   method call(t1: Ref, t2: Ref)
10  requires acc(UniqueT(t1), write) &&
    acc(SharedT(t1), wildcard)
11  requires acc(UniqueT(t2), write) &&
    acc(SharedT(t2), wildcard)
12  ensures acc(UniqueT(t1), write) &&
    acc(SharedT(t1), wildcard)
13  ensures acc(SharedT(t2), wildcard)
14  {
15      uniqueBorrowedParam(t1)
16      uniqueBorrowedParam(t2)
17      uniqueParam(t2)
18  }
```

Listing 54: Function call with unique parameter encoding

6.4.2 Functions with Shared Parameters

When functions with a shared parameter are called, their encoding may require the addition of `inhale` and `exhale` statements. The annotation system allows functions with shared parameters to be called by passing unique references. However, the function's conditions alone are not sufficient to properly encode these calls.

For example, passing a unique reference to a function expecting a shared (non-borrowed) parameter will result in the loss of uniqueness for that reference, which is encoded by exhaling the unique predicate. Similarly, when a unique reference is passed to a function expecting a borrowed-shared parameter, the uniqueness is preserved, but any field of that reference can be modified. This is encoded by exhaling and then re-inhaling the unique predicate of that reference.

Figure 3 summarizes the `inhale` and `exhale` statements added during the encoding of a function call.
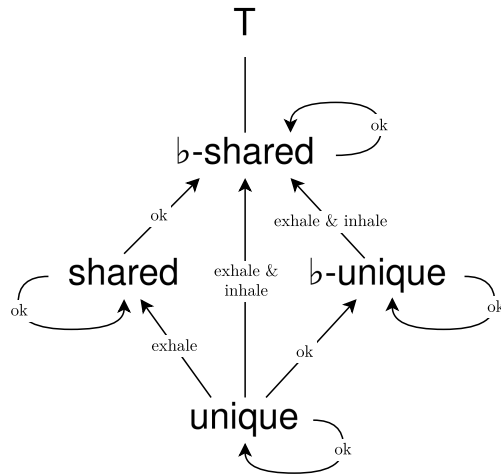
T

♭-shared   ok

ok   exhale & inhale

shared   exhale & inhale   ♭-unique

ok   ok

exhale   ok

unique   ok

Figure 3: Extra statements added for functions call encoding

```kotlin
1   fun sharedParam(        Kotlin
2       t: T
3   ) {
4   }
5
6   fun sharedBorrowedParam(
7       @Borrowed t: T
8   ) {
9   }
10
11  fun call(@Unique t: T) {
12      sharedBorrowedParam(t)
13      sharedParam(t)
14  }
```

```viper
1   method sharedParam(t: Ref)     Viper
2   requires acc(SharedT(t), wildcard)
3   ensures acc(SharedT(t), wildcard)
4
5   method sharedBorrowedParam(t: Ref)
6   requires acc(SharedT(t), wildcard)
7   ensures acc(SharedT(t), wildcard)
8
9   method call(t: Ref)
10  requires acc(UniqueT(t), write) &&
        acc(SharedT(t), wildcard)
11  ensures acc(SharedT(t), wildcard)
12  {
13    exhale acc(UniqueT(t), write)
14    sharedBorrowedParam(t)
15    inhale acc(UniqueT(t), write)
16
17    exhale acc(UniqueT(t), write)
18    sharedParam(t)
19  }
```

Listing 55: Function call with shared parameter encoding

6.5 Stack Example

Finally, Listing 56 shows how the example from Section 4.3 is encoded in Viper. In this example, shared predicates are omitted for readability, as they would be empty. Moreover, the `UniqueAny` predicate does not add additional value to the encoding. However, it can be replaced with any class predicate without affecting the correctness of the encoding.

```
1    field value: Ref, next: Ref, root: Ref                        🐏 Viper
2
3    predicate UniqueAny(this: Ref)
4
5    predicate UniqueNode(this: Ref) {
6        acc(this.value) && (this.value != null ==> UniqueAny(this.value)) &&
7        acc(this.next) && (this.next != null ==> UniqueNode(this.next))
8    }
9
10   predicate UniqueStack(this: Ref) {
11       acc(this.root) && (this.root != null ==> UniqueNode(this.root))
12   }
13
14   method constructorNode(val: Ref, nxt: Ref) returns (res: Ref)
15   requires val != null ==> UniqueAny(val)
16   requires nxt != null ==> UniqueNode(nxt)
17   ensures UniqueNode(res)
18   ensures unfolding UniqueNode(res) in res.value == val && res.next == nxt
19
20   method push(this: Ref, val: Ref)
21   requires UniqueStack(this)
22   requires val != null ==> UniqueAny(val)
23   ensures UniqueStack(this) {
24       var r: Ref
25       unfold UniqueStack(this)
26       r := this.root
27       this.root := constructorNode(val, r)
28       fold UniqueStack(this)
29   }
30
31   method pop(this: Ref) returns (res: Ref)
32   requires UniqueStack(this)
33   ensures UniqueStack(this)
34   ensures res != null ==> UniqueAny(res) {
35       var val: Ref
36       unfold UniqueStack(this)
37       if(this.root == null) { val := null }
38       else {
39           unfold UniqueNode(this.root)
40           val := this.root.value
41           this.root := this.root.next
42       }
43       fold UniqueStack(this)
44       res := val
45   }
```

Listing 56: Stack encoding in Viper

Chapter 7

Conclusion

7.1 Results

This thesis has introduced a novel uniqueness system for the Kotlin language, bringing several important improvements over existing approaches [2,7,33]. The system provides improved flexibility in managing field accesses, particularly in handling nested properties within Kotlin programs. It allows the correct permissions for nested field accesses to be determined at any point of the program, without imposing any restrictions based on whether properties are unique, shared, or inaccessible. Furthermore, the uniqueness of properties can evolve during program execution, similarly to variables. The system also introduces a clear distinction between borrowed-shared and borrowed-unique references, making it easier to integrate uniqueness annotations into existing codebases. Indeed, one of its key benefits is the ability to be adopted incrementally, enabling developers to incorporate it into their Kotlin code without the need for significant changes.

The uniqueness system has been rigorously formalized, detailing the rules and constraints necessary to ensure that unique references are properly maintained within a program.

Finally, this work has demonstrated how the uniqueness system can be used to encode Kotlin into Viper more precisely, enabling more accurate and reliable verification of Kotlin programs.

7.2 Future Work

7.2.1 Extending the Language

Extending the range of Kotlin features supported by the annotation system is a natural next step for this work.

One area for extension is support for `while` loops. Currently, loops are not well supported by SnaKt due to the lack of support for inferring invariants. As a result, handling loops was not a primary focus for the uniqueness system.

Lambdas are another important feature in Kotlin that the uniqueness system must support. Lambdas often capture references through closures, which presents challenges for maintaining uniqueness. Handling these references correctly requires careful tracking to ensure that the captured variables do not lead to unintended aliasing. Bao et al. [5] have proposed a system for tracking aliasing in higher-order functional programs, which could provide valuable insights for addressing these challenges.

7.2.2 Improving Borrowed Fields Flexibility

Currently, fields of borrowed parameters are subject to restrictions necessary for ensuring system soundness when unique references are passed to functions expecting shared borrowed parameters. Specifically, borrowed fields can only be reassigned using a unique reference. However, in some cases, allowing reassignment with shared references would also be safe. Similarly, borrowed fields become inaccessible after be-

ing read, even though there are situations where they could safely remain shared. Introducing rules to manage these scenarios would enhance the system's flexibility in handling borrowed fields, representing a significant improvement.

7.2.3 Tracking of Local Aliases

The uniqueness system proposed by Zimmerman et al. [33] guarantees the following uniqueness invariant: "A unique object is stored at most once on the heap. In addition, all usable references to a unique object from the local environment are precisely inferred." This invariant allows for the creation of local aliases of unique objects without compromising their uniqueness.

In contrast, the uniqueness system proposed in this work takes a different approach. When local aliases are created, the original reference becomes inaccessible, and the local alias is treated as unique. This design choice prioritizes flexibility in the usage of paths while maintaining simplicity in the typing rules.

However, there is potential for future improvements to the system. By refining the existing rules, it may be possible to achieve a uniqueness invariant that allows the creation of controlled aliases without losing the guarantees of uniqueness. Such an enhancement would expand the range of Kotlin code supported by the system while preserving the integrity of uniqueness guarantees.

7.2.4 Checking Annotations

This work presents a uniqueness system and shows how it can be used to verify Kotlin code by encoding it into Viper. Currently, SnaKt assumes that any annotated Kotlin program is well-typed according to the typing rules presented in Chapter 5.

To improve the system, a static checker is under development. This checker will use Kotlin's control flow graph to ensure that the annotations satisfy the typing rules of the uniqueness system. By integrating this static analysis, SnaKt will start to encode Kotlin into Viper only if the program is well-typed, reducing the need for manual validation and increasing the reliability of the verification process.

7.2.5 Proving the Soundness of the Annotation System

Another area for future work is proving the soundness of the proposed annotation system. Establishing soundness would involve formally demonstrating that the system's rules and annotations prevent illegal aliasing and correctly track ownership throughout program execution. For instance, it would be important to prove that when a path is unique at any given point in the program, no other accessible paths point to the same object as that path. Additionally, it would be valuable to demonstrate that borrowed parameters are not further aliased by any function, ensuring that the borrowing mechanism preserves the integrity of reference uniqueness and prevents unintended aliasing.

Bibliography

[1] Marat Akhin and Mikhail Belyaev. 2021. Kotlin language specification. *Kotlin Language Specification* (2021).

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. 2002. Alias annotations for program understanding. *ACM SIGPLAN Notices* 37, 11 (2002), 311–330.

[3] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods (14th International Symposium)*, 2022. Springer, 88–108. Retrieved from https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5

[4] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2019. ACM, 1–30. https://doi.org/10.1145/3360573

[5] Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA (October 2021). https://doi.org/10.1145/3485516

[6] Robert L Bocchino Jr. 2013. Alias control for deterministic parallelism. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification* (2013), 156–195.

[7] John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience* 31, 6 (2001), 533–553.

[8] Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). 2013. *Aliasing in Object-Oriented Programming: types, analysis, and verification*. Springer-Verlag, Berlin, Heidelberg.

[9] Dave Clarke, James Noble, and Tobias Wrigstad. 2013. Beyond the geneva convention on the treatment of object aliasing. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification* (2013), 1–6.

[10] ETH Zürich Department of Computer Science. Viper Language. Retrieved from https://www.pm.inf.ethz.ch/research/viper.html

[11] Marco Eilers and Peter Müller. 2018. Nagini: a static verifier for Python. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30*, 2018. 596–603.

[12] Marco Eilers, Malte Schwerhoff, and Peter Müller. 2024. Verification Algorithms for Automated Separation Logic Verifiers. In *International Conference on Computer Aided Verification*, 2024. 362–386.

[13] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/https://doi.org/10.1016/0304-3975(87)90045-4

[14] Dana Harrington. 2006. Uniqueness logic. *Theoretical Computer Science* 354, 1 (2006), 24–41.

[15] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. 2013. Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, 2013. Springer, 451–476. Retrieved from https://doi.org/10.1007/978-3-642-39038-8_19

[16] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. 1992. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.* 3, 2 (April 1992), 11–16. https://doi.org/10.1145/130943.130947

[17] Apple Inc. and the Swift project authors. Swift Parameter Modifiers. Retrieved from https://docs.swift.org/swift-book/documentation/the-swift-programming-language/declarations/#Parameter-Modifiers

[18] Apple Inc. and the Swift project authors. Swift Ownership Manifesto. Retrieved from https://github.com/swiftlang/swift/blob/main/docs/OwnershipManifesto.md

[19] Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an assertion language for mutable data structures. *SIGPLAN Not.* 36, 3 (January 2001), 14–26. https://doi.org/10.1145/373243.375719

[20] JetBrains. 2024. SnaKt: a Formal Verification Plugin for Kotlin. Retrieved from https://github.com/jesyspa/kotlin/tree/formal-verification/plugins/formal-verification

[21] JetBrains. 2024. Kotlin Programming Language. Retrieved from https://kotlinlang.org/

[22] JetBrains. 2024. Kotlin docs - Shared mutable state and concurrency. Retrieved from https://kotlinlang.org/docs/shared-mutable-state-and-concurrency.html

[23] Ralf Jung. 2020. Understanding and evolving the Rust programming language. (2020).

[24] Daniel Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *Programming Languages and Systems*, 2022. Springer International Publishing, Cham, 346–375.

[25] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS)*, 2016. Springer-Verlag, 41–62. Retrieved from https://doi.org/10.1007/978-3-662-49122-5_2

[26] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution. In *Computer Aided Verification (CAV) (LNCS)*, 2016. Springer-Verlag, 405–425. Retrieved from http://link.springer.com/chapter/10.1007/978-3-319-41528-4_22

[27] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP (July 2019). https://doi.org/10.1145/3341714

[28] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *Computer Science Logic: 15th International Workshop, CSL 2001 10th Annual Conference of the EACSL Paris, France, September 10–13, 2001, Proceedings 15*, 2001. 1–19.

[29] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[30] Dmitry Savvinov. 2019. Kotlin Contracts. Retrieved from https://github.com/Kotlin/KEEP/blob/master/proposals/kotlin-contracts.md

[31] The Rust Team. 2024. The Rust Programming Language. Retrieved from https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html

[32] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification (CAV) (LNCS)*, 2021. Springer International Publishing, 367–379. Retrieved from https://link.springer.com/chapter/10.1007/978-3-030-81685-8_17

[33] Conrad Zimmerman, Catarina Gamboa, Alcides Fonseca, and Jonathan Aldrich. 2023. Latte: Lightweight Aliasing Tracking for Java. *arXiv preprint arXiv:2309.05637* (2023).

Appendix A

Typing Rules

All the following rules are to be considered for a given program P, where fields within the same class, as well as across different classes, must have distinct names.

A.1 General

$$\text{M-Type-1} \ \frac{m(x_0 : \alpha_0\beta_0, ..., x_n : \alpha_n\beta_n) : \alpha\{\text{begin}_m; s; \text{return}_m e\} \in P}{\text{m-type}(m) = \alpha_0\beta_0, ..., \alpha_n\beta_n \rightarrow \alpha}$$

$$\text{M-Type-2} \ \frac{m(x_0 : \alpha_0\beta_0, ..., x_n : \alpha_n\beta_n) : \alpha \in P}{\text{m-type}(m) = \alpha_0\beta_0, ..., \alpha_n\beta_n \rightarrow \alpha}$$

$$\text{M-Args-1} \ \frac{m(x_0 : \alpha_0\beta_0, ..., x_n : \alpha_n\beta_n) : \alpha\{\text{begin}_m; s; \text{return}_m e\} \in P}{\text{args}(m) = x_0, ..., x_n}$$

$$\text{M-Args-2} \ \frac{m(x_0 : \alpha_0\beta_0, ..., x_n : \alpha_n\beta_n) : \alpha \in P}{\text{args}(m) = x_0, ..., x_n}$$

$$\text{F-Default} \ \frac{\text{class } C\left(\overline{f' : \alpha'_f}, f : \alpha_f, \overline{f'' : \alpha''_f}\right) \in P}{\text{default}(f) = \alpha_f}$$

A.2 Well-Formed Contexts

$$\text{Not-In-Base} \ \frac{}{p \notin \cdot} \qquad\qquad \text{Not-In-Rec} \ \frac{p \neq p' \qquad p \notin \Delta}{p \notin (p' : \alpha\beta, \Delta)}$$

$$\text{Ctx-Base} \ \frac{}{\cdot \ \text{ctx}} \qquad\qquad \text{Ctx-Rec} \ \frac{\Delta \ \text{ctx} \qquad p \notin \Delta}{p : \alpha\beta, \Delta \ \text{ctx}}$$

A.3 Sub-Paths and Super-Paths

A.3.1 Definition

$$\text{Sub-Path-Base} \ \frac{}{p \sqsubset p.f} \qquad\qquad \text{Sub-Path-Rec} \ \frac{p \sqsubset p'}{p \sqsubset p'.f}$$

$$\text{Sub-Path-Eq-1} \frac{}{p \sqsubseteq p} \qquad\qquad\qquad \text{Sub-Path-Eq-2} \frac{p \sqsubset p'}{p \sqsubseteq p'}$$

A.3.2 Remove

$$\text{Remove-Empty} \frac{}{\cdot \setminus p = \cdot} \qquad\qquad\qquad \text{Remove-Base} \frac{}{(p : \alpha\beta, \Delta) \setminus p = \Delta}$$

$$\text{Remove-Rec} \frac{\Delta \setminus p = \Delta' \qquad p \neq p'}{(p' : \alpha\beta, \Delta) \setminus p = p' : \alpha\beta, \Delta'}$$

A.3.3 Deep Remove

$$\text{Deep-Remove-Empty} \frac{}{\cdot \ominus p = \cdot}$$

$$\text{Deep-Remove-Discard} \frac{p \sqsubseteq p' \qquad \Delta \ominus p = \Delta'}{(p' : \alpha\beta, \Delta) \ominus p = \Delta'}$$

$$\text{Deep-Remove-Keep} \frac{p \not\sqsubseteq p' \qquad \Delta \ominus p = \Delta'}{(p' : \alpha\beta, \Delta) \ominus p = (p' : \alpha\beta, \Delta')}$$

A.3.4 Replace

$$\text{Replace} \frac{\Delta \ominus p = \Delta'}{\Delta[p \mapsto \alpha\beta] = \Delta', p : \alpha\beta}$$

A.3.5 Get Super-Paths

$$\text{Get-Super-Paths-Empty} \frac{}{\cdot \vdash \text{superPaths}(p) = \cdot}$$

$$\text{Get-Super-Paths-Discard} \frac{\neg(p \sqsubset p') \qquad \Delta \vdash \text{superPaths}(p) = p_0 : \alpha_0\beta_0, ..., p_n : \alpha_n\beta_n}{p' : \alpha\beta, \Delta \vdash \text{superPaths}(p) = p_0 : \alpha_0\beta_0, ..., p_n : \alpha_n\beta_n}$$

$$\text{Get-Super-Paths-Keep} \frac{p \sqsubset p' \qquad \Delta \vdash \text{superPaths}(p) = p_0 : \alpha_0\beta_0, ..., p_n : \alpha_n\beta_n}{p' : \alpha\beta, \Delta \vdash \text{superPaths}(p) = p' : \alpha\beta, p_0 : \alpha_0\beta_0, ..., p_n : \alpha_n\beta_n}$$

A.4 Relations between Annotations

A.4.1 Partial Ordering

$$\text{Rel-Id} \frac{}{\alpha\beta \preccurlyeq \alpha\beta} \qquad\qquad\qquad \text{Rel-Trans} \frac{\alpha\beta \preccurlyeq \alpha'\beta' \qquad \alpha'\beta' \preccurlyeq \alpha''\beta''}{\alpha\beta \preccurlyeq \alpha''\beta''}$$

$$\text{Rel-Shared-}\flat \frac{}{\text{shared } \flat \preccurlyeq \top} \qquad\qquad \text{Rel-Shared} \frac{}{\text{shared} \preccurlyeq \text{shared } \flat}$$

$$\text{Rel-Unique-}\flat \frac{}{\text{unique } \flat \preccurlyeq \text{shared } \flat} \qquad\qquad \text{Rel-Unique-1} \frac{}{\text{unique} \preccurlyeq \text{shared}}$$

$$\text{Rel-Unique-2} \frac{}{\text{unique} \preccurlyeq \text{unique } \flat}$$

A.4.2 Passing

$$\text{Pass-}\flat \frac{\alpha\beta \preccurlyeq \alpha'\flat}{\alpha\beta \rightsquigarrow \alpha'\flat \rightsquigarrow \alpha\beta} \qquad\qquad \text{Pass-Unique} \frac{}{\text{unique} \rightsquigarrow \text{unique} \rightsquigarrow \top}$$

$$\text{Pass-Shared} \frac{\alpha \preccurlyeq \text{shared}}{\alpha \rightsquigarrow \text{shared} \rightsquigarrow \text{shared}}$$

A.5 Paths

A.5.1 Root

$$\text{Root-Base} \frac{}{\text{root}(x) = x} \qquad\qquad \text{Root-Rec} \frac{\text{root}(p) = x}{\text{root}(p.f) = x}$$

A.5.2 Lookup

$$\text{Lookup-Base} \frac{(p : \alpha\beta, \Delta) \text{ ctx}}{(p : \alpha\beta, \Delta)\langle p \rangle = \alpha\beta} \qquad\qquad \text{Lookup-Rec} \frac{\begin{array}{c}(p : \alpha\beta, \Delta) \text{ ctx} \\ p \neq p' \qquad \Delta\langle p' \rangle = \alpha'\beta'\end{array}}{(p : \alpha\beta, \Delta)\langle p' \rangle = \alpha'\beta'}$$

$$\text{Lookup-Default} \frac{\text{default}(f) = \alpha}{\cdot\langle p.f \rangle = \alpha}$$

A.5.3 Get

$$\text{Get-Var} \frac{\Delta\langle x \rangle = \alpha\beta}{\Delta(x) = \alpha\beta} \qquad\qquad \text{Get-Path} \frac{\Delta(p) = \alpha\beta \qquad \Delta\langle p.f \rangle = \alpha'}{\Delta(p.f) = \bigsqcup\{\alpha\beta, \alpha'\}}$$

A.5.4 Standard Form

$$\text{Std-Empty} \frac{}{\cdot \vdash \text{std}(p, \alpha\beta)} \qquad\qquad \text{Std-Rec-1} \frac{\neg(p \sqsubset p') \qquad \Delta \vdash \text{std}(p, \alpha\beta)}{p' : \alpha\beta, \Delta \vdash \text{std}(p, \alpha\beta)}$$

$$\text{Std-Rec-2} \frac{\begin{array}{c}p \sqsubset p' \qquad \text{root}(p) = x \\ (x : \alpha\beta)(p') = \alpha''\beta'' \qquad \alpha'\beta' \preccurlyeq \alpha''\beta'' \qquad \Delta \vdash \text{std}(p, \alpha\beta)\end{array}}{p' : \alpha'\beta', \Delta \vdash \text{std}(p, \alpha\beta)}$$

A.6 Unification

A.6.1 Pointwise LUB

$$\text{Ctx-LUB-Empty} \frac{}{\cdot \ \sqcup \ \cdot \ = \ \cdot} \qquad\qquad \text{Ctx-LUB-Sym} \frac{}{\Delta_1 \sqcup \Delta_2 = \Delta_2 \sqcup \Delta_1}$$

$$\text{Ctx-LUB-1} \frac{\Delta_2\langle p\rangle = \alpha''\beta'' \qquad \Delta_2 \setminus p = \Delta_2' \\ \Delta_1 \sqcup \Delta_2' = \Delta' \qquad \bigsqcup\{\alpha\beta, \alpha''\beta''\} = \alpha'\beta'}{(p:\alpha\beta, \Delta_1) \sqcup \Delta_2 = p:\alpha'\beta', \Delta'}$$

$$\text{Ctx-LUB-2} \frac{x \notin \Delta_2 \qquad \Delta_1 \sqcup \Delta_2 = \Delta'}{(x:\alpha\beta, \Delta_1) \sqcup \Delta_2 = x:\top, \Delta'}$$

A.6.2 Removal of Local Declarations

$$\text{Remove-Locals-Base} \frac{}{\cdot \blacktriangleleft \Delta = \cdot}$$

$$\text{Remove-Locals-Keep} \frac{\text{root}(p) = x \qquad \Delta_1\langle x\rangle = \alpha'\beta' \qquad \Delta \blacktriangleleft \Delta_1 = \Delta'}{p:\alpha\beta, \Delta \blacktriangleleft \Delta_1 = p:\alpha\beta, \Delta'}$$

$$\text{Remove-Locals-Discard} \frac{\text{root}(p) = x \qquad x \notin \Delta_1 \qquad \Delta \blacktriangleleft \Delta_1 = \Delta'}{p:\alpha\beta, \Delta \blacktriangleleft \Delta_1 = \Delta'}$$

A.6.3 Unify

$$\text{Unify} \frac{\Delta_1 \sqcup \Delta_2 = \Delta_{\sqcup} \qquad \Delta_{\sqcup} \blacktriangleleft \Delta = \Delta'}{\text{unify}(\Delta; \Delta_1; \Delta_2) = \Delta'}$$

A.7 Normalization

$$\text{N-Empty} \frac{}{\text{normalize}(\cdot) = \cdot}$$

$$\text{N-Rec} \frac{\bigsqcup(\alpha_i\beta_i \mid p_i = p_0 \wedge 0 \le i \le n) = \alpha_{\sqcup}\beta_{\sqcup} \\ \text{normalize}(p_i : \alpha_i\beta_i \mid p_i \ne p_0 \wedge 0 \le i \le n) = p_0' : \alpha_0'\beta_0', ..., p_m' : \alpha_m'\beta_m'}{\text{normalize}(p_0 : \alpha_0\beta_0, ..., p_n : \alpha_n\beta_n) = p_0 : \alpha_{\sqcup}\beta_{\sqcup}, p_0' : \alpha_0'\beta_0', ..., p_m' : \alpha_m'\beta_m'}$$

A.8 Statements Typing

$$\text{Begin} \frac{\text{m-type}(m) = \alpha_0\beta_0, ..., \alpha_n\beta_n \to \alpha \qquad \text{args}(m) = x_0, ..., x_n}{\cdot \vdash \text{begin}_m \dashv x_0 : \alpha_0\beta_0, ..., x_n : \alpha_n\beta_n}$$

$$\text{Seq} \frac{\Delta \vdash s_1 \dashv \Delta_1 \qquad \Delta_1 \vdash s_2 \dashv \Delta'}{\Delta \vdash s_1 ; s_2 \dashv \Delta'} \qquad\qquad \text{Decl} \frac{x \notin \Delta}{\Delta \vdash \text{var } x \dashv \Delta, x : \top}$$

$$\text{Call} \frac{\begin{array}{c} \forall 0 \leq i \leq n : \Delta(p_i) = \alpha_i \beta_i \\ \text{m-type}(m) = \alpha_0^m, \beta_0^m, ..., \alpha_n^m \beta_n^m \to \alpha_r \\ \forall 0 \leq i \leq n : \Delta \vdash \text{std}(p_i, \alpha_i^m \beta_i^m) \\ \forall 0 \leq i, j \leq n : \left(i \neq j \wedge p_i = p_j \right) \Rightarrow \alpha_i^m = \text{shared} \\ \forall 0 \leq i, j \leq n : p_i \sqsubset p_j \Rightarrow \left(\Delta(p_j) = \text{shared} \vee \alpha_i^m = \alpha_j^m = \text{shared} \right) \\ \Delta \ominus (p_0, ..., p_n) = \Delta' \qquad \forall 0 \leq i \leq n : \alpha_i \beta_i \rightsquigarrow \alpha_i^m \beta_i^m \rightsquigarrow \alpha_i' \beta_i' \\ \text{normalize}(p_0 : \alpha_0' \beta_0', ..., p_n : \alpha_n' \beta_n') = p_0' : \alpha_0'' \beta_0'', ..., p_m' : \alpha_m'' \beta_m'' \end{array}}{\Delta \vdash m(p_0, ..., p_n) \dashv \Delta', p_0' : \alpha_0'' \beta_0'', ..., p_m' : \alpha_m'' \beta_m''}$$

$$\text{Assign-Null} \frac{\Delta(p) = \alpha\beta \qquad \Delta[p \mapsto \text{unique}] = \Delta'}{\Delta \vdash p = \text{null} \dashv \Delta'}$$

$$\text{Assign-Call} \frac{\begin{array}{c} \Delta(p) = \alpha' \beta' \qquad \Delta \vdash m(\overline{p}) \dashv \Delta_1 \\ \text{m-type}(m) = \alpha_0 \beta_0, ..., \alpha_n \beta_n \to \alpha \\ (\beta' = \flat) \Rightarrow (\alpha = \text{unique}) \qquad \Delta_1[p \mapsto \alpha] = \Delta' \end{array}}{\Delta \vdash p = m(\overline{p}) \dashv \Delta'}$$

$$\text{Assign-Unique} \frac{\begin{array}{c} p' \not\sqsubseteq p \qquad \Delta(p) = \alpha\beta \qquad \Delta(p') = \text{unique} \qquad \Delta[p' \mapsto \top] = \Delta_1 \\ \Delta \vdash \text{superPaths}(p') = p'.\overline{f_0} : \alpha_0 \beta_0, ..., p'.\overline{f_n} : \alpha_n \beta_n \qquad \Delta_1[p \mapsto \text{unique}] = \Delta' \end{array}}{\Delta \vdash p = p' \dashv \Delta', p.\overline{f_0} : \alpha_0 \beta_0, ..., p.\overline{f_n} : \alpha_n \beta_n}$$

$$\text{Assign-Shared} \frac{\begin{array}{c} p' \not\sqsubseteq p \qquad \Delta(p) = \alpha \qquad \Delta(p') = \text{shared} \\ \Delta \vdash \text{superPaths}(p') = p'.\overline{f_0} : \alpha_0 \beta_0, ..., p'.\overline{f_n} : \alpha_n \beta_n \qquad \Delta[p \mapsto \text{shared}] = \Delta' \end{array}}{\Delta \vdash p = p' \dashv \Delta', p.\overline{f_0} : \alpha_0 \beta_0, ..., p.\overline{f_n} : \alpha_n \beta_n}$$

$$\text{Assign-}\flat\text{-Field} \frac{\begin{array}{c} p'.f \not\sqsubseteq p \qquad \Delta(p) = \alpha\beta \qquad \Delta(p'.f) = \alpha' \flat \\ \alpha' \neq \top \qquad (\beta = \flat) \Rightarrow (\alpha' = \text{unique}) \qquad \Delta[p'.f \mapsto \top] = \Delta_1 \\ \Delta \vdash \text{superPaths}(p'.f) = p'.f.\overline{f_0} : \alpha_0 \beta_0, ..., p'.f.\overline{f_n} : \alpha_n \beta_n \qquad \Delta_1[p \mapsto \alpha'] = \Delta' \end{array}}{\Delta \vdash p = p'.f \dashv \Delta', p.\overline{f_0} : \alpha_0 \beta_0, ..., p.\overline{f_n} : \alpha_n \beta_n}$$

$$\text{If} \frac{\begin{array}{c} \Delta(p_1) \neq \top \qquad \Delta(p_2) \neq \top \\ \Delta \vdash s_1 \dashv \Delta_1 \qquad \Delta \vdash s_2 \dashv \Delta_2 \qquad \text{unify}(\Delta; \Delta_1; \Delta_2) = \Delta' \end{array}}{\Delta \vdash \text{if } p_1 == p_2 \text{ then } s_1 \text{ else } s_2 \dashv \Delta'}$$

$$\text{Return-p} \ \dfrac{\begin{array}{ccc} \text{m-type}(m) = \alpha_0^m, \beta_0^m, ..., \alpha_n^m \beta_n^m \to \alpha_r & \Delta(p) = \alpha\beta & \alpha\beta \preccurlyeq \alpha_r \\ \Delta \vdash \text{std}(p, \alpha_r) & \forall 0 \le i, j \le n : (\alpha_i \beta_i \ne \text{unique}) \Rightarrow \Delta \vdash \text{std}(p_i, \alpha_i \beta_i) \end{array}}{\Delta \vdash \text{return}_m \ p \dashv \cdot}$$