Uniqueness Deep Dive

Uniqueness annotations for Kotlin

Francesco Protopapa
Intern in Kotlin Formal Verification

Importance of Controlling Aliasing

“Aliasing is one of the key features of object-oriented programming languages, but it is both a blessing and a curse.”

Aliasing in Object-Oriented Programming - Preface

Proving functional properties

Lack of aliasing control complicates the validation of basic functional properties

```
class A(var x: Boolean)

fun f(a1: A, a2: A): Boolean {
    contract { returns(true) }
    a1.x = true
    a2.x = false
    return a1.x
}
```

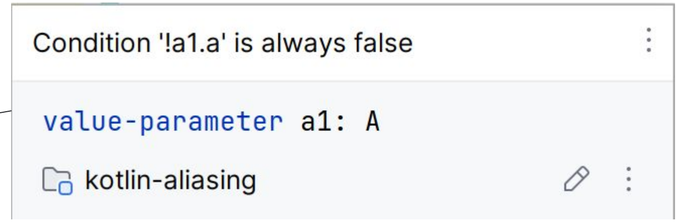
If 'a1' and 'a2' are aliased, the function returns false

```
class A(var x: Boolean)

fun f(a1: A): Boolean {
    contract { returns(true) }
    a1.x = true
    return a1.x
}
```

In a concurrent context, 'a1.x' may change before being returned

Static Analysis in IntelliJ

```kotlin
class A(var a: Boolean = false)

fun f(a1: A, a2: A) {
    a1.a = true
    a2.a = false
    if (!a1.a) {
        println("ALIASED!")
    }
}

fun main() {
    val a1 = A()
    f(a1, a1)
}
```



Prints “ALIASED!”

Uniqueness and smart casts

How guarantees on uniqueness can improve smart casts

```
class A(var x: Boolean?)

fun f(a: A) {
    if (a.x ≠ null) {
        a.x = !a.x
    }
}
```


Smart cast to 'Boolean' is impossible, because 'a.x' is a mutable property that could have been changed by this time

```
class A(var x: Boolean?)

fun f(a: @Unique A) {
    if (a.x ≠ null) {
        a.x = !a.x
    }
}
```


If we are sure that 'a' is not aliased, it is safe to cast 'a.x' to 'Boolean'

An annotation system for aliasing control

Goals

- Improve verification in our plugin
- Annotations that are easy to understand
- Already existing code can be annotated gradually
- Provide annotations for variables and **class members**

Annotation system

$$\alpha ::= \text{unique} \mid \text{shared} \mid \top$$

α annotations

- **Fields**, **return values, parameters** and **receiver** are annotated with **unique** or **shared**
- $\top$ can only be inferred,
- A reference annotated with **T** is not accessible
- A reference annotated with **unique** when accessed
  - is **null**
  - is the sole accessible reference pointing to that object
- **shared** does not provide guarantees on uniqueness

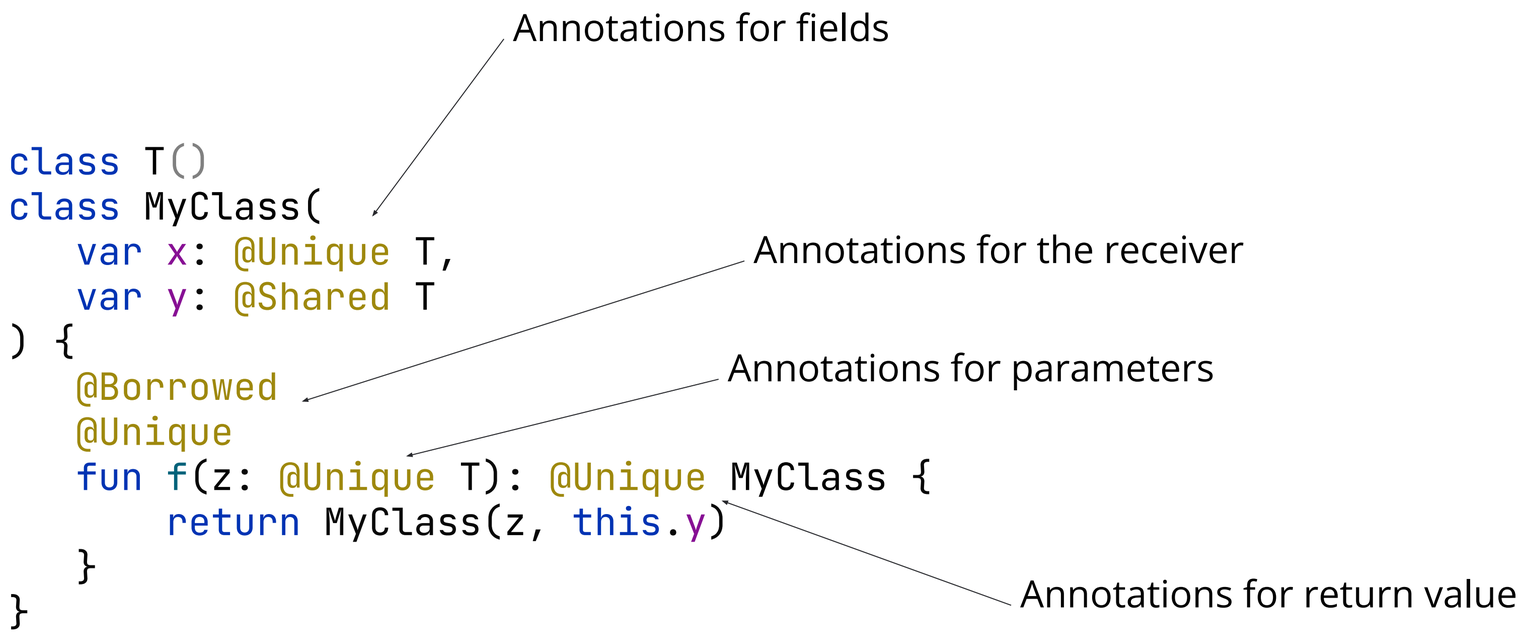
Annotation system

$$\beta ::= \cdot \mid \flat$$

β annotations

- $\flat$ stands for **borrowed**
- Only function **parameters** and **receiver** can be annotated with β
- If a parameter is **borrowed** no new aliases may be added to the heap

How annotated code looks like




```
class T()
class MyClass(
    var x: @Unique T,
    var y: @Shared T
) {
    @Borrowed
    @Unique
    fun f(z: @Unique T): @Unique MyClass {
        return MyClass(z, this.y)
    }
}
```

Formalization

$$\mathrm{CL} ::= \texttt{class}\ C\left(\overline{f : \alpha_f}\right)$$

$$\alpha_f ::= \mathrm{unique} \mid \mathrm{shared}$$

$$\beta ::= \cdot \mid \flat$$

$$M ::= m\left(\overline{\alpha_f \beta\ x}\right) : \alpha_f\{\mathrm{begin}_m; \overline{s}; \mathrm{return}_m e\}$$

$$p ::= x \mid p.f$$

$$e ::= \mathrm{null} \mid p \mid m(\overline{p})$$

$$s ::= \mathrm{var}\ x \mid p = e \mid \mathrm{if}\ p_1 == p_2\ \mathrm{then}\ \overline{s_1}\ \mathrm{else}\ \overline{s_2} \mid m(\overline{p})$$

```
class C(var x: @Unique Any?, var y: @Unique Any?)

@Unique | @Shared

@Borrowed

fun f (x :@Borrowed @Unique Any?, y :@Shared Any?)
  : @Unique C { ... }

x | x.y | x.y.z | ...

null | x.y.z | f(x, x.y, ...)

var x | x.y = null | if (x == x.y){ ... } else { ... }
```

Formalization

Grammar

$$\mathrm{CL} ::= \texttt{class}\ C\left(\overline{f : \alpha_f}\right)$$

$$\alpha_f ::= \mathrm{unique} \mid \mathrm{shared}$$

$$\beta ::= \cdot \mid \flat$$

$$M ::= m\left(\overline{\alpha_f \beta\ x}\right) : \alpha_f\{\mathrm{begin}_m; \overline{s}; \mathrm{return}_m e\}$$

$$p ::= x \mid p.f$$

$$e ::= \mathrm{null} \mid p \mid m(\overline{x})$$

$$s ::= \mathrm{var}\ x \mid p = e \mid \mathrm{if}\ p_1 == p_2\ \mathrm{then}\ \overline{s_1}\ \mathrm{else}\ \overline{s_2}$$

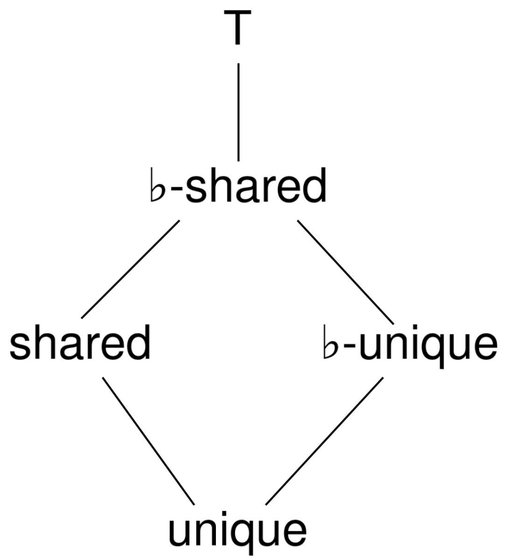
Context

$$\alpha ::= \mathrm{unique} \mid \mathrm{shared} \mid \top$$

$$\beta ::= \cdot \mid \flat$$

$$\Delta ::= \cdot \mid p : \alpha\beta, \Delta$$

Relations between Annotations



passed variable — signature — result

$$\alpha_1\beta_1 \rightsquigarrow \alpha_2\beta_2 \rightsquigarrow \alpha_3\beta_3$$

$$\text{Pass-Bor}\ \frac{\alpha\beta \preccurlyeq \alpha'\flat}{\alpha\beta \rightsquigarrow \alpha'\flat \rightsquigarrow \alpha\beta}$$

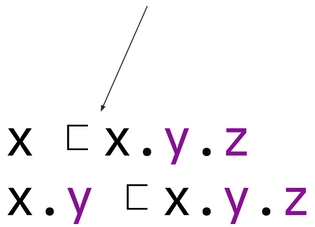
$$\text{Pass-Un}\ \frac{}{\text{unique} \rightsquigarrow \text{unique} \rightsquigarrow \top}$$

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

```
class T()

fun bu(x: @Borrowed @Unique T) {}
fun bs(x: @Borrowed @Shared T) {}
fun u(x: @Unique T) {}
fun s(x: @Shared T) {}

fun test() {
    var x = T()
//      Δ = x: unique
    bu(x)
//      Δ = x: unique
    bs(x)
//      Δ = x: unique
    u(x)
//      Δ = x: T
    x = T()
//      Δ = x: unique
    s(x)
//      Δ = x: shared
}
```

Paths

- Contexts can also contain paths
- If a path is not in the context, the annotation in the declaration is considered
- The actual permissions of a path is determined by the LUB of the paths included (⊏) in it



Context

$$\alpha ::= \text{unique} \mid \text{shared} \mid \top$$
$$\beta ::= \cdot \mid \flat$$
$$\Delta ::= \cdot \mid p : \alpha\beta, \Delta$$

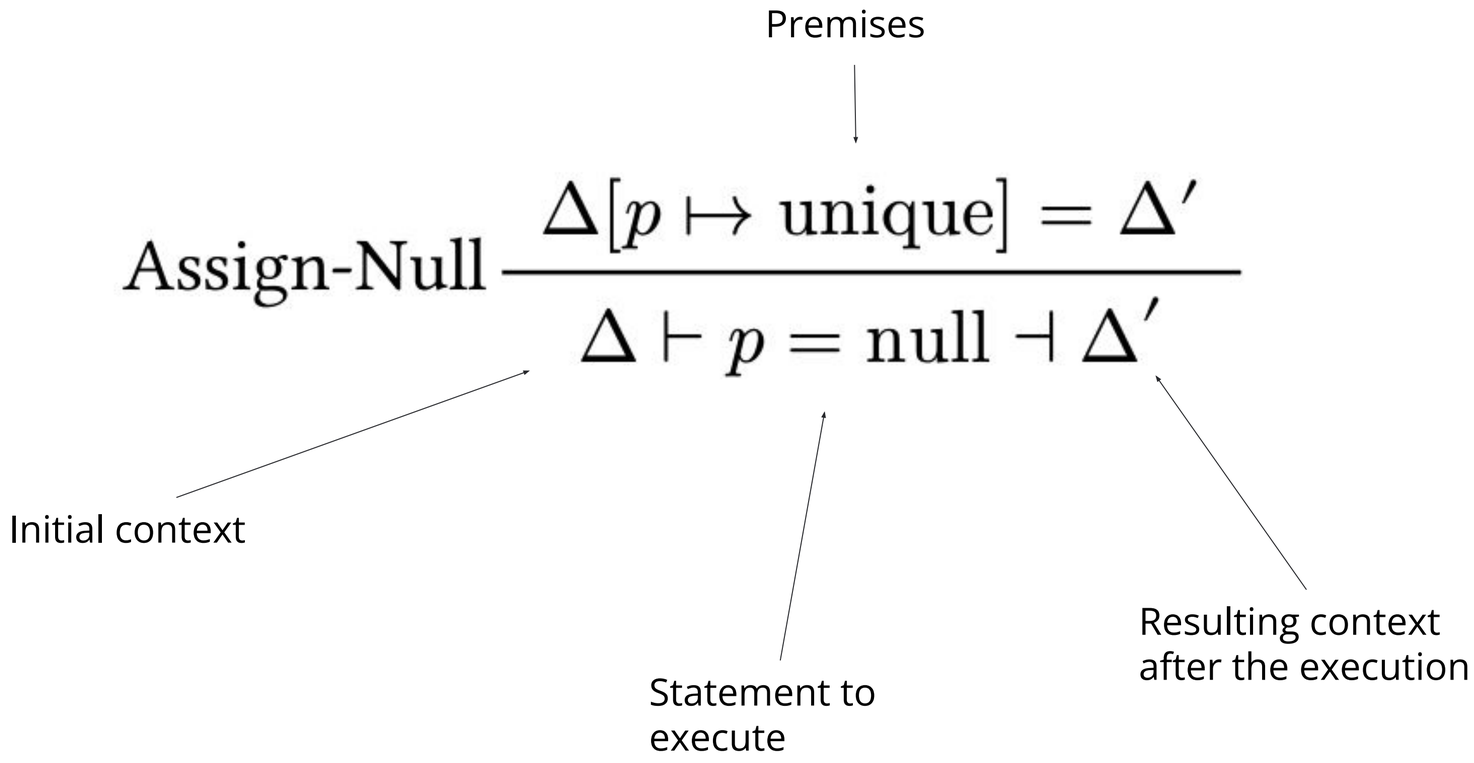
```
class A()
class B(var z: @Unique A)
class C(var y: @Shared B)

fun f(x: @Unique C) {
    var a = x.y.z
}
```

LUB {unique, shared, unique} = shared

Statement typing

Statement typing

Introduction to the notation



Statement typing

Basic rules

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

```
class A()
fun main(){
    // Δ = .
    var a: A?
    // Δ = a: T
    a = null
    // Δ = a: unique
}
```

Context without p

Context without paths rooted in p

$$\text{Replace}\ \frac{\Delta \setminus p = \Delta_1 \qquad \Delta_1 \ominus p = \Delta'}{\Delta[p \mapsto \alpha\beta] = \Delta', p : \alpha\beta}$$

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

Statement typing

Assignments

For now assignments like 'p.f = p' are not allowed

p has to be in the context

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

p’ has to be unique

Paths rooted in p’

p and p’ are re-assigned, paths rooted in p and p’ are not present in Δ’

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

Paths originally rooted in p’ are rooted in p in the resulting context

Statement typing

Assignments

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

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

```
class T()
class A(var f: @Unique T = T())

fun f(x: @Unique A) {
//     Δ = x: unique
    var y: A
//     Δ = x: unique, y: T
    y = A()
//     Δ = x: unique, y: unique
    x.f = y.f
//     Δ = x: unique, y: unique, y.f: T
    y = A()
//     Δ = x: unique, y: unique
}
```

Statement typing

If statements and unification

- Variables not in Δ (locally declared in the branches) will not be in unify(Δ ; Δ_1 ; Δ_2)
- For the remaining variables, the unification will contain the LUB between Δ_1 and Δ_2

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

```
class T()
class A(var f: @Unique T)

fun consumeUnique(x: @Unique T){}
fun consumeShared(x: @Shared A){}

fun f(a: @Unique A, b: Boolean) {
//     Δ = a: unique
   if (b){
       consumeUnique(a.f)
//         Δ1 = a: unique, a.f: T
   } else {
       consumeShared(a)
//         Δ2 = a: shared
   }
//     unify(Δ; Δ1; Δ2)
//       = a: LUB{unique, shared},
//         a.f: LUB{T, shared}
//     Δ = a: shared, a.f: T
}
```

Statement typing

Method call

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

Statement typing

Method call

$\forall 0 \leq i \leq n : \Delta(p_i) = \alpha_i \beta_i$ → p_i is in the context with permissions $\alpha_i \beta_i$

$$\text{Call}\ \frac{}{\Delta \vdash m(p_0, ..., p_n) \dashv \Delta', p'_0 : \alpha''_0 \beta''_0, ..., p'_m : \alpha''_m \beta''_m}$$

Statement typing

Method call

Type of the method

$$\text{m-type}(m) = \alpha_0^m, \beta_0^m, ..., \alpha_n^m \beta_n^m \to \alpha_r \beta_r$$

$$\forall 0 \leq i \leq n : \Delta \vdash \text{std}(p_i, \alpha_i^m)$$

All the paths rooted in p_i have the expected permissions

$$\text{Call}\ \frac{}{\Delta \vdash m(p_0, ..., p_n) \dashv \Delta', p_0' : \alpha_0'' \beta_0'', ..., p_m' : \alpha_m'' \beta_m''}$$

```
class T()
class A(var f: @Unique T)

fun f(a: @Unique A) {}
fun use_f(x: @Unique A, y: @Unique A) {
    // Δ = x: Unique, y: Unique
    y.f = x.f
    // Δ = x: Unique, y: Unique, x.f: T
    f(x) // error
}
```

'x.f' does not have std permissions when 'x' is passed

Statement typing

Method call

$$\text{m-type}(m) = \alpha_0^m, \beta_0^m, ..., \alpha_n^m \beta_n^m \to \alpha_r \beta_r$$

$$\forall 0 \leq i, j \leq n : \left(i \neq j \wedge p_i = p_j\right) \Rightarrow \alpha_i^m = \text{shared}$$

$$\text{Call}\ \frac{}{\Delta \vdash m(p_0, ..., p_n) \dashv \Delta', p_0' : \alpha_0'' \beta_0'', ..., p_m' : \alpha_m'' \beta_m''}$$

```
class A()

fun f1(x: @Shared A, y: @Shared A) {}
fun f2(x: @Unique A, y: @Shared A) {}

fun use_f1(x: @Unique A) {
//     Δ = x: Unique
   f1(x, x) // ok
//     Δ = x: Shared
}

fun use_f2(x: @Unique A) {
//     Δ = x: Unique
   f2(x, x) // error
}
```

'x' is passed more than once but is also expected to be unique

Statement typing

Method call

Fields of an object that has been passed to a method can be passed too, but only if the nested one is shared or they are both expected to be shared.

$$\text{m-type}(m) = \alpha_0^m, \beta_0^m, ..., \alpha_n^m \beta_n^m \to \alpha_r \beta_r$$

$$\forall 0 \leq i, j \leq n : p_i \sqsubset p_j \Rightarrow \Big(\Delta(p_j) = \text{shared} \vee a_i^m = a_j^m = \text{shared}\Big)$$

$$\text{Call}\ \frac{}{\Delta \vdash m(p_0, ..., p_n) \dashv \Delta', p_0' : \alpha_0'' \beta_0'', ..., p_m' : \alpha_m'' \beta_m''}$$

```
class T()
class B(var f: @Unique T)

fun f(x: @Shared B, y: @Shared T) {}
fun use_f(x: @Unique B) {
//    Δ = x: Unique
   f(x, x.f) // ok
//    Δ = x: Shared, x.f: Shared
}

fun g(x: @Unique B, y: @Shared T) {}
fun use_g(b: @Unique B) {
//    Δ = b: Unique
   g(b, b.f) // error
}
```

'b.f' cannot be passed since 'b' is passed as Unique and Δ(b.f) = Unique

Statement typing

Method call

$$\forall 0 \leq i \leq n : \Delta(p_i) = \alpha_i \beta_i$$
$$\text{m-type}(m) = \alpha_0^m, \beta_0^m, ..., \alpha_n^m \beta_n^m \to \alpha_r \beta_r$$

$$\text{Call} \frac{\Delta \setminus (p_0, ..., p_n) = \Delta' \qquad \forall 0 \leq i \leq n : \alpha_i \beta_i \rightsquigarrow \alpha_i^m \beta_i^m \rightsquigarrow \alpha_i' \beta_i'}{\Delta \vdash m(p_0, ..., p_n) \dashv \Delta', p_0' : \alpha_0'' \beta_0'', ..., p_m' : \alpha_m'' \beta_m''}$$

Statement typing

Method call

$$\forall 0 \leq i \leq n : \Delta(p_i) = \alpha_i \beta_i$$
$$\text{m-type}(m) = \alpha_0^m, \beta_0^m, ..., \alpha_n^m \beta_n^m \to \alpha_r \beta_r$$

$$\text{Call}\ \frac{\Delta \setminus (p_0, ..., p_n) = \Delta' \qquad \forall 0 \leq i \leq n : \alpha_i \beta_i \rightsquigarrow \alpha_i^m \beta_i^m \rightsquigarrow \alpha_i' \beta_i' \qquad \text{normalize}(p_0 : \alpha_0' \beta_0', ..., p_n : \alpha_n' \beta_n') = p_0' : \alpha_0'' \beta_0'', ..., p_m' : \alpha_m'' \beta_m''}{\Delta \vdash m(p_0, ..., p_n) \dashv \Delta', p_0' : \alpha_0'' \beta_0'', ..., p_m' : \alpha_m'' \beta_m''}$$

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

```
fun f(x: @Borrowed @Shared T, y: @Shared T) {}

fun use_f(x: @Unique T) {
// Δ = x: unique
   f(x, x)
// Δ = normalize(x: unique, x: shared) = x: shared
}
```

Statement typing

Assign call

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

Guarantees on return

- Borrowed parameters haven’t been aliased and are still std
- If return value is annotated unique, the returned expression has to be unique and std
- If return value is annotated shared, the returned expression has to be shared and std

Next steps

- Formal rules for guarantees on return
- Decide what happens to members of a borrowed variable
- Use these annotations in our plugin
- Annotations checking
- While loops
- Lambdas
- Tracking local aliases

Thank you!

Francesco Protopapa