1. Formalization

Grammar

$$
\begin{aligned}
\text{CL} &::= \texttt{class}\ C\left(\overline{f : \alpha_f}\right) \\
\alpha_f &::= \text{unique} \mid \text{shared} \\
\beta &::= \cdot \mid \flat \\
M &::= m\left(\overline{\alpha_f \beta\ x}\right) : \alpha_f \{\text{begin}_m; \overline{s}; \text{return}_m e\} \\
p &::= x \mid p.f \\
e &::= \text{null} \mid p \mid m(\overline{p}) \\
s &::= \text{var}\ x \mid p = e \mid \text{if}\ p_1 == p_2\ \text{then}\ \overline{s_1}\ \text{else}\ \overline{s_2} \mid m(\overline{p}) \\
&\quad \mid \text{while}\ p_1 == p_2\ \text{do}\ \overline{s}
\end{aligned}
$$

Context

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

- Only **fields**, **method parameters**, and **return values** have to be annotated.
- A reference annotated as `unique` may either be `null` or point to an object, and it is the sole **accessible** reference pointing to that object.
- A reference marked as `shared` can point to an object without being the exclusive reference to that object.
- `T` is an annotation that can only be inferred and means that the reference is **not accessible**.
- $\flat$ (borrowed) indicates that the function receiving the reference won't create extra aliases to it, and on return, its fields will maintain at least the permissions stated in the class declaration.
- Annotations on fields indicate only the default permissions, in order to understand the real permissions of a fields it is necessary to look at the context. This concept is formalized by rules in Section 1.5 and shown in Listing 1.
- Primitive fields are not considered
- `this` can be seen as a parameter
- constructors can be seen as functions returning a `unique` value

1.1. General

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

1.2. Context

- The same variable/field cannot appear more than once in a context.
- Contexts are always **finite**
- If not present in the context, fields have a default annotation that is the one written in the class declaration

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

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

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

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

$$\text{Lookup-Default}\ \frac{\mathrm{type}(p) = C \qquad \texttt{class}\ C\left(\overline{f' : \alpha'_f}, f : \alpha, \overline{f'' : \alpha''_f}\right)}{\cdot\ \langle p.f \rangle = \alpha}$$

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

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

1.3. SubPaths

If $p_1 \sqsubset p_2$ holds, we say that

- p_1 is a **sub**-path of p_2
- p_2 is a **sup**-path of p_1

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

$$\text{SubPath-Eq-1}\ \frac{p = p'}{p \sqsubseteq p'} \qquad \text{SubPath-Eq-2}\ \frac{p \sqsubset p'}{p \sqsubseteq p'}$$

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

$$\text{Remove-SupPathsEq-Keep}\ \frac{p \not\sqsubseteq p' \qquad \Delta \ominus p = \Delta'}{(p' : \alpha\beta, \Delta) \ominus p = (p' : \alpha\beta, \Delta')} \qquad \text{Replace}\ \frac{\Delta \ominus p = \Delta'}{\Delta[p \mapsto \alpha\beta] = \Delta', p : \alpha\beta}$$

$$\text{Get-SupPaths-Empty}\ \frac{}{\cdot \vdash \mathrm{supPaths}(p) = \cdot}$$

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

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

1.4. Annotations relations

- $\alpha\beta \preccurlyeq \alpha'\beta'$ means that $\alpha\beta$ can be passed where $\alpha'\beta'$ is expected.

- $\alpha\beta \rightsquigarrow \alpha'\beta' \rightsquigarrow \alpha''\beta''$ means that after passing a reference annotated with $\alpha\beta$ as argument where $\alpha'\beta'$ is expected, the reference will be annotated with $\alpha''\beta''$ right after the method call.

$$\text{A-Id}\ \frac{}{\alpha\beta \preccurlyeq \alpha\beta} \qquad \text{A-Trans}\ \frac{\alpha\beta \preccurlyeq \alpha'\beta' \qquad \alpha'\beta' \preccurlyeq \alpha''\beta''}{\alpha\beta \preccurlyeq \alpha''\beta''} \qquad \text{A-Bor-Sh}\ \frac{}{\text{shared}\ \flat \preccurlyeq \top}$$

$$\text{A-Sh}\ \frac{}{\text{shared} \preccurlyeq \text{shared}\ \flat} \qquad \text{A-Bor-Un}\ \frac{}{\text{unique}\ \flat \preccurlyeq \text{shared}\ \flat} \qquad \text{A-Un-1}\ \frac{}{\text{unique} \preccurlyeq \text{shared}}$$

$$\text{A-Un-2}\ \frac{}{\text{unique} \preccurlyeq \text{unique}\ \flat} \qquad \text{Pass-Bor}\ \frac{\alpha\beta \preccurlyeq \alpha'\flat}{\alpha\beta \rightsquigarrow \alpha'\flat \rightsquigarrow \alpha\beta} \qquad \text{Pass-Un}\ \frac{}{\text{unique} \rightsquigarrow \text{unique} \rightsquigarrow \top}$$

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

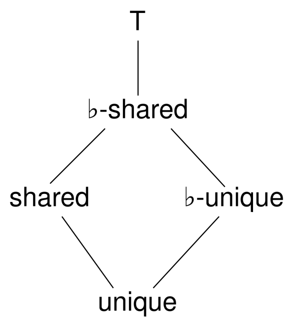

Figure 1: Lattice obtained by annotations relations rules

1.5. Paths

- $\bigsqcup\{\alpha_0\beta_0, ..., \alpha_n\beta_n\}$ identifies the least upper bound of the annotations based on the lattice in Figure 1.
- Note that even if $p.f$ is annotated as unique in the class declaration, $\Delta(p.f)$ can be shared (or $\top$) if $\Delta(p) = \text{shared}$ (or $\top$)
- Note that fields of a borrowed parameter are borrowed too and they need to be treated carefully in order to avoid unsoundness. Specifically, borrowed fields:
  - Can be passed as arguments to other functions (if relation rules are respected).
  - Have to become `T` after being read (even if shared).
  - Can only be reassigned with a `unique`.
- Note that $(\Delta(p) = \alpha\beta) \Rightarrow (\Delta\langle\text{root}(p)\rangle = \alpha'\beta')$ i.e. the root is present in the context.
- $\Delta \vdash \text{std}(p, \alpha\beta)$ means that paths rooted in p have the right permissions when passing p where $\alpha\beta$ is expected. To understand better why these rules are necessary look at the example in Listing 2.
- Note that in the rule "Std-Rec-2" the premise $(x : \alpha\beta)(p') = \alpha''\beta''$ means that the evaluation of p' in a context in which there is only $x : \alpha\beta$ is $\alpha''\beta''$

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

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

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

1.6. Unification

- $\Delta_1 \sqcup \Delta_2$ is the pointwise lub of Δ_1 and Δ_2.
  - If a variable x is present in only one context, it will be annotated with $\top$ in $\Delta_1 \sqcup \Delta_2$.
  - If a path $p.f$ is missing in one of the two contexts, we can just consider the annotation in the class declaration.
- $\Delta \blacktriangleleft \Delta_1$ is used to maintain the correct context when exiting a scope.
  - Δ represents the resulting context of the inner scope.
  - Δ_1 represents the context at the beginning of the scope.
  - The result of the operation is a context where paths rooted in variable locally declared inside the scope are removed.
- $\text{unify}(\Delta; \Delta_1; \Delta_2)$ means that we want to unify Δ_1 and Δ_2 starting from a parent environment Δ.
  - A path p contained in Δ_1 or Δ_2 such that $\text{root}(p) = x$ is not contained Δ will not be included in the unfication.
  - The annotation of variables contained in the unfication is the least upper bound of the annotation in Δ_1 and Δ_2.

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

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

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

$$\text{Ctx-Lub-3}\ \frac{\Delta_2\langle x\rangle = \alpha''\beta'' \qquad \Delta_2 \setminus x = \Delta_2' \qquad \Delta_1 \sqcup \Delta_2' = \Delta' \qquad \bigsqcup\{\alpha\beta, \alpha''\beta''\} = \alpha'\beta'}{(x : \alpha\beta, \Delta_1) \sqcup \Delta_2 = x : \alpha'\beta', \Delta'}$$

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

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

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

1.7. Normalization

- Normalization takes a list of annotated p and retruns a list in which duplicates are substituted with the least upper bound.
- Normalization is required for method calls in which the same variable is passed more than once.

```
fun f(x: ♭ shared, y: shared)
fun use_f(x: unique) {
  // Δ = x: unique
  f(x, x)
  // Δ = normalize(x: unique, x:shared) = x: shared
}
```

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

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

1.8. Statements

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

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

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

$$\text{Seq-Base}\ \frac{}{\Delta \vdash \overline{s} \equiv \cdot \dashv \Delta}$$

$$\text{Seq-Rec}\ \frac{\Delta \vdash s_0 \dashv \Delta_1 \qquad \Delta_1 \vdash \overline{s'} \dashv \Delta'}{\Delta \vdash \overline{s} \equiv s_0; \overline{s'} \dashv \Delta'}$$

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

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

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

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

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

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

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

Note: Since they can be easily desugared, there are no rules for returnning `null` or a method call.

- `return null` $\equiv$ `var fresh ; fresh = null ; return fresh`
- `return m(...)` $\equiv$ `var fresh ; fresh = m(...) ; return fresh`

The same can be done for the guard of if statements:

- `if (p1 == null) ...` $\equiv$ `var p2 ; p2 = null ; if(p1 == p2) ...`
- `if (p1 == m(...)) ...` $\equiv$ `var p2 ; p2 = m(...) ; if(p1 == p2) ...`

2. Examples

2.1. Paths-permissions:

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

fun use_a(a: @Unique A)

fun f1(a: @Shared A){
  //  Δ = a: Shared
  // ==>  Δ(a.f) = shared even if `f` is annotated ad unique
}

fun f2(a: @Unique A){
  //  Δ = a: Unique
  // ==>  Δ(a.f) = Unique
  use_a(a)
  // after passing `a` to `use_a` it can no longer be accessed
  // Δ = a: T
  // Δ(a.f) = T even if `f` is annotated ad unique
}
```

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

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 standard permissions when 'x' is passed
}
```

2.2. Call premises explaination:

- $\forall 0 \leq i \leq n : \Delta \vdash \mathrm{std}(p_i, \alpha_i^m \beta_i^m)$:
  We need all the arguments to have at least standard permissions for their fields.
- $\forall 0 \leq i, j \leq n : \left(i \neq j \land p_i = p_j\right) \Rightarrow \alpha_i^m = \mathrm{shared}$:
  If the same variable/field is passed more than once, it can only be passed where shared is expected.

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

fun f1(x: @Unique A, y: @Borrowed @Shared A)
fun f2(x: @Borrowed @Shared A, y: @Borrowed @Shared A)
fun f3(x: @Shared A, y: @Borrowed @Shared A)

fun use_f1(x: @Unique A){
  // Δ = x: Unique
  f1(x, x) // error: 'x' is passed more than once but is also expected to be
unique
}

fun use_f2_f3(x: @Unique A){
  // Δ = x: Unique
  f2(x, x) // ok, uniqueness is also preserved since both the args are borrowed
  // Δ = x: Unique
  f3(x, x) // ok, but uniqueness is lost since one of the args is not borrowed
  // Δ = x: Shared
}
```

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

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

fun f1(x: @Unique A, y: @Shared C) {}
fun use_f1(x: @Unique A) {
// Δ = x: Unique
    f1(x, x.f) // ok
// Δ = x: T, x.f: Shared
// Note that even if x.f is marked shared in the context,
// it is not accessible since Δ(x.f) = T
}

fun f2(x: @Unique B, y: @Shared C) {}
fun use_f2(b: @Unique B) {
// Δ = b: Unique
    f2(b, b.f) // error: 'b.f' cannot be passed since 'b' is passed as Unique and
Δ(b.f) = Unique
// Δ(b.f) = Unique
// It is correct to raise an error since f2 expects x.f to be unique
}

fun f3(x: @Shared B, y: @Shared C) {}
fun use_f3(x: @Unique B) {
// Δ = x: Unique
    f3(x, x.f) // ok
// Δ = x: Shared, x.f: Shared
}
```

2.3. Borrowed fields

Fields of a borrowed variable are borrowed too. But differently from variables, they can be read/written and so these operation require special rules.

- `Assign-Borrowed-Field` tells us what happens when reading a borrowed field. The most important thing to notice is that after being read, the field will be annotated with $\top$, even if it is shared. Doing this is necessary to guarantee soundness while passing unique variables to functions expecting a borrowed shared argument.
- For the same reason, borrowed fields can only be re-assigned to something that is unique. Otherwise passing a unique to a function expecting a borrowed shared argument cannot guarantee that uniqueness is preserved.

```
class A(var n: Int)
class B(var a: @Unique A)

fun f(b: @Shared @Borrowed B) {
//    Δ = b : Shared Borrowed
    var temp = b.a
//    Δ = b : Shared Borrowed, b.a : T, temp: Shared

    // before returning, b needs to be in std form
    // but at this point it can only be re-assigned with a unique
    // re-assigning b.a with a shared would cause unsoundness to a caller passing
a unique

    b.a = A(0)
//    Δ = b : Shared Borrowed, b.a : Unique, temp: Shared

    // now the function can return with no problems
}

fun use_f(b: @Unique B) {
//    Δ = b : Unique
    // also Δ(b.f) = Unique
    f(b)
//    Δ = b : Unique
    // moreover it is guaranteed that also Δ(b.a) = Unique
}
```

2.4. Smart casts

Since if-statements guards cannot create new aliases, having a variable or a field access in the guard will not modify its uniqueness.

```
class T
class A(var t: @Unique T?)

fun use_t(t : @Shared @Borrowed T){

}

fun f(a1: @Unique @Borrowed A, a2 : @Shared @Borrowed A) {
//    Δ = a1 : Unique Borrowed, a2 : Shared Borrowed
    if (a1.t != null) {
    //    Δ = a1 : Unique Borrowed, a2 : Shared Borrowed
        use_t(a1.t) // here it is safe to smart cast because Δ(a1.t) = Unique
(Borrowed)
    //    Δ = a1 : Unique Borrowed, a2 : Shared Borrowed
    }

//    Δ = a1 : Unique Borrowed, a2 : Shared Borrowed
    if(a2.t != null){
    //    Δ = a1 : Unique Borrowed, a2 : Shared Borrowed
        use_t(a2.t!!) // here it is NOT safe to smart cast because Δ(a2.t) =
Shared (Borrowed)
    //    Δ = a1 : Unique Borrowed, a2 : Shared Borrowed
    }
}
```

2.5. Stack example

This shows how the example presented in LATTE paper works with this system.

```
class Node(var value: @Unique Any?, var next: @Unique Node?)

class Stack(var root: @Unique Node?) {
    @Borrowed @Unique
    fun push(value: @Unique Any?) {
        // Δ = this: borrowed unique, value: unique

        val r = this.root
        // Δ = this: borrowed unique, this.root: T, value: unique, r: unique

        val n = Node(value, r)
        // Δ = this: borrowed unique, this.root: T, value: T, r: T, n: unique

        this.root = n
        // Δ = this: borrowed unique, this.root: unique, value: T, r: T, n: T
    }

    @Borrowed @Unique
    fun pop(): @Unique Any? {
        // Δ = this: borrowed unique

        val value: Any?
        // Δ = this: borrowed unique, value: T
        if (this.root == null) {
            value = null
            // Δ = this: borrowed unique, value: unique
        } else {
            value = this.root.value // Note: smart cast 'this.root' to Node
            // Δ = this: borrowed unique, this.root.value: T, value: unique

            val next = this.root.next  // Note: smart cast 'this.root' to Node
            // Δ = this: borrowed unique, this.root.value: T, this.root.next: T,
value: unique, next: unique

            this.root = next
          // Δ = this: borrowed unique, this.root: unique, value: unique, next: T

            // Note: doing this.root = this.root.next works too
        }
        // Unification...
        // Δ = this: borrowed unique, this.root: unique, value: unique
        return value
    }
}
```