subtyping

```swift
class Cat {
  func makeNoise() {
    print("meow!")
  }
}

class Dog {
  func makeNoise() {
    print("woof!")
  }
}
```

```swift
let cats: [Cat] = [Cat(), Cat()]
let dogs: [Dog] = [Dog(), Dog()]

cats.map { (c) in c.makeNoise() }
dogs.map { (d) in d.makeNoise() }
```

```
let harmony = [Cat(), Dog()]
```

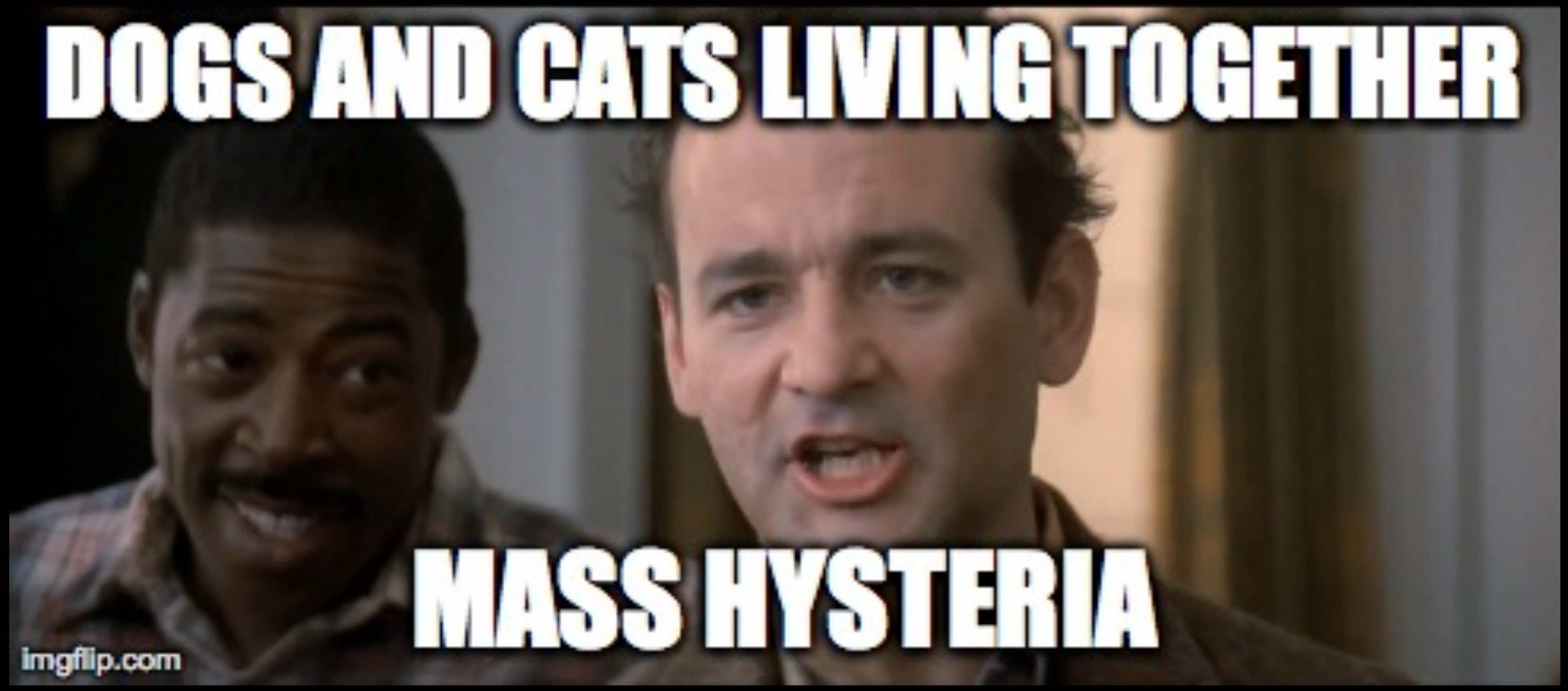
```
let harmony = [Cat(), Dog()]

// harmony: [?]
```

```
let harmony = [Cat(), Dog()]

// harmony.map { (a) in a.makeNoise() }
```

```
let harmony = [Cat(), Dog()]
```



```swift
class Animal {
  func makeNoise() {}
}

class Cat: Animal {
  override func makeNoise() {
    print("meow!")
  }
}

class Dog: Animal {
  override func makeNoise() {
    print("woof!")
  }
}
```

```
let harmony: [Animal] = [Cat(), Dog()]

harmony.map { (a) in a.makeNoise() }
```

```swift
func sayHi(_ animal: Animal) {
  print("hi!")
  animal.makeNoise()
}

sayHi(Cat())

sayHi(Dog())
```

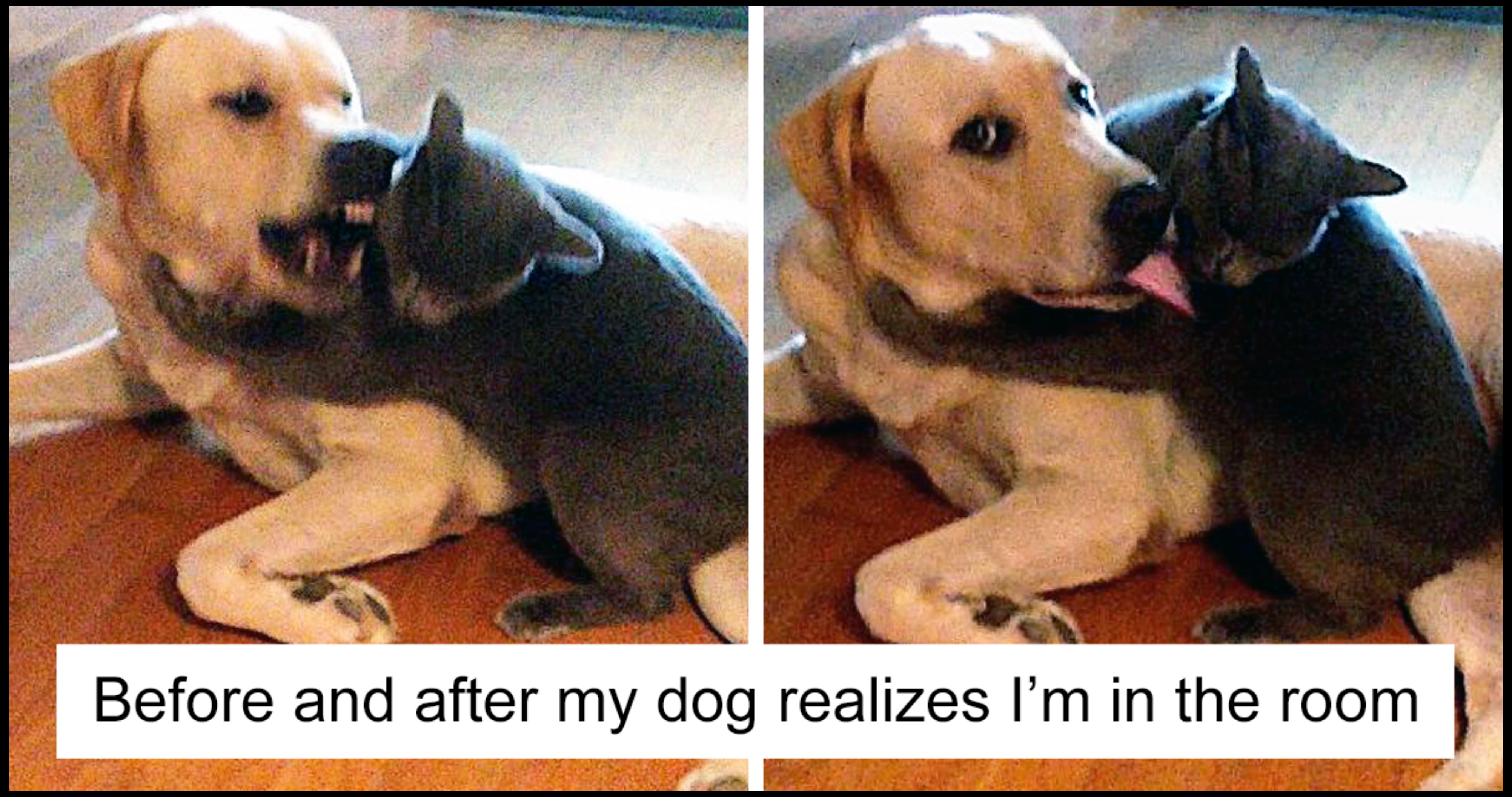
```swift
protocol Noisy {
  func makeNoise()
}

protocol Pettable {
  func pet()
}

class Cat: Noisy, Pettable {
  func makeNoise() {
    print("meow!")
  }
  func pet() {
    self.makeNoise()
  }
}
```

```swift
let harmony: [Any] = [Cat(), Dog()]
```

```
let harmony: [Any] = [Cat(), Dog()]
```



what do these have in common?

subsumption

if e has type σ

and σ is a subtype of τ

then e has type τ

subsumption

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau}$$

```
type big = { x: t1, y: t2, z: t3 }
type small = { x: t1, y: t2 }

fun f (s : small) = #x s
val b = { x = 1, y = 2, z = 3 }

(* f b *)
```

width

$$\frac{}{\left\{\, l_i : \tau_i^{\;i \in 1..n+k} \right\} <: \left\{\, l_i : \tau_i^{\;i \in 1..n} \right\}}$$

depth

$$\frac{\sigma_i <: \tau_i}{\left\{ l_i : \sigma_i{}^{i \in 1..n} \right\} <: \left\{ l_i : \tau_i{}^{i \in 1..n} \right\}}$$

variant width

$$\frac{}{\left\langle l_i : \tau_i{}^{i \in 1..n} \right\rangle <: \left\langle l_i : \tau_i{}^{i \in 1..n+k} \right\rangle}$$

$$\frac{}{\left\langle l_i : \tau_i{}^{i \in 1..n} \right\rangle <: \left\langle l_i : \tau_i{}^{i \in 1..n+k} \right\rangle}$$

$$\frac{}{\left\{ l_i : \tau_i{}^{i \in 1..n+k} \right\} <: \left\{ l_i : \tau_i{}^{i \in 1..n} \right\}}$$

$$\frac{}{\left\langle l_i : \tau_i^{\;i\in 1..n} \right\rangle <: \left\langle l_i : \tau_i^{\;i\in 1..n+k} \right\rangle}$$

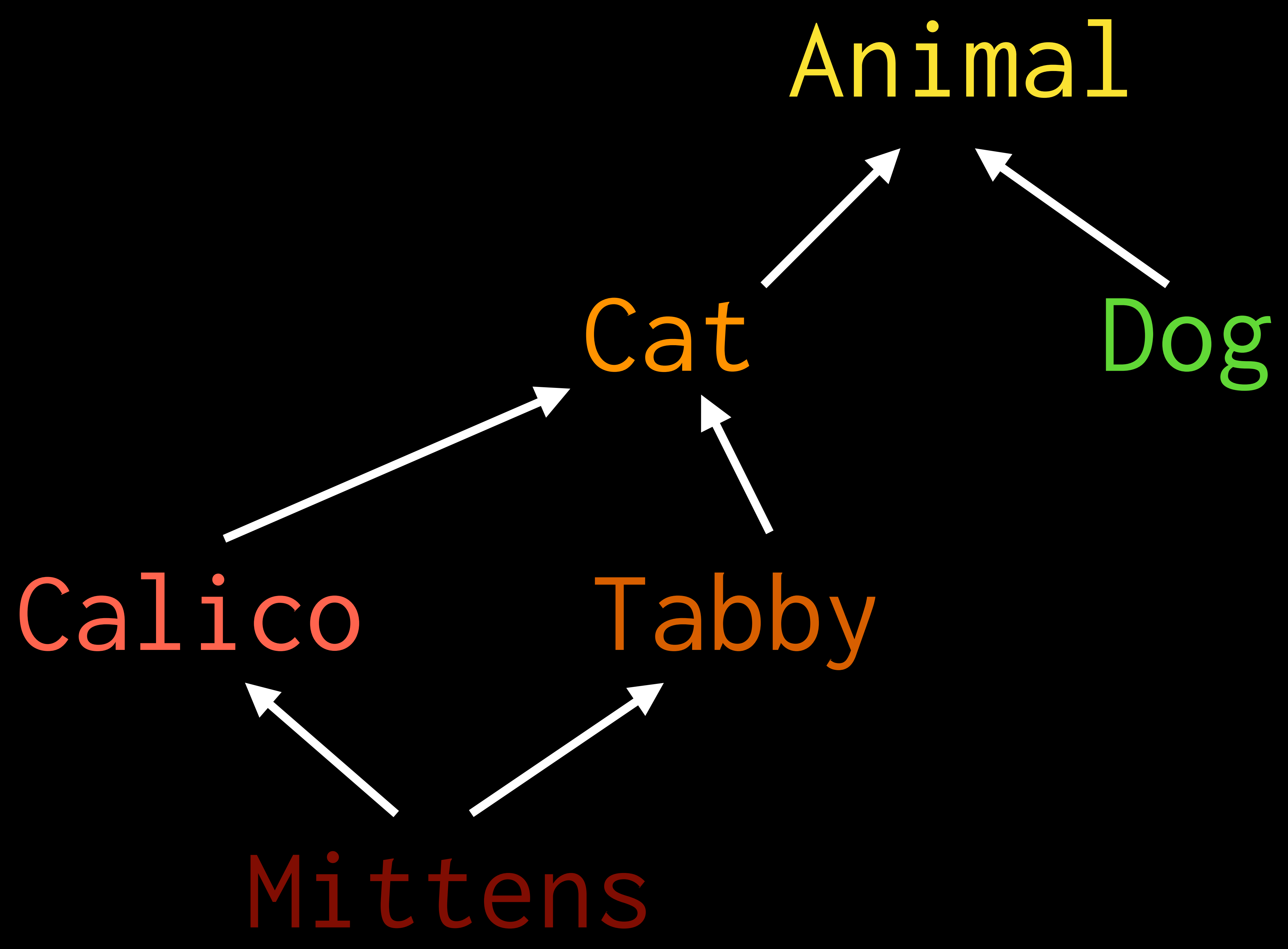
$$\frac{}{\left\{ l_i : \tau_i^{\;i\in 1..n+k} \right\} <: \left\{ l_i : \tau_i^{\;i\in 1..n} \right\}}$$

```
datatype big =
  Xb of t1
| Yb of t2
| Zb of t3

datatype small =
  Xs of t1
| Ys of t2
```

small product big sum

big product small sum


Animal
Cat
Dog
Calico
Tabby
Mittens

Complex

Real

Rational

Long

Int

List Bool

Maybe Bool

Bool

Int Bool Cat Dog Socket Image List AbstractProxyFactorySingletonBean

Enum Animal Container

Int Bool Cat Dog Socket Image List AbstractProxyFactorySingletonBean

Enum Animal Container

Int Bool Cat Dog Socket Image List AbstractProxyFactorySingletonBean

TCPSocket LinkedList

Copyable Synchronized

Enum Animal Container

Int Bool Cat Dog Socket Image List AbstractProxyFactorySingletonBean

TCPSocket LinkedList

Any

Copyable Synchronized

Enum Animal Container

Int Bool Cat Dog Socket Image List AbstractProxyFactorySingletonBean

TCPSocket LinkedList

Any

Copyable Synchronized

Enum Animal Container

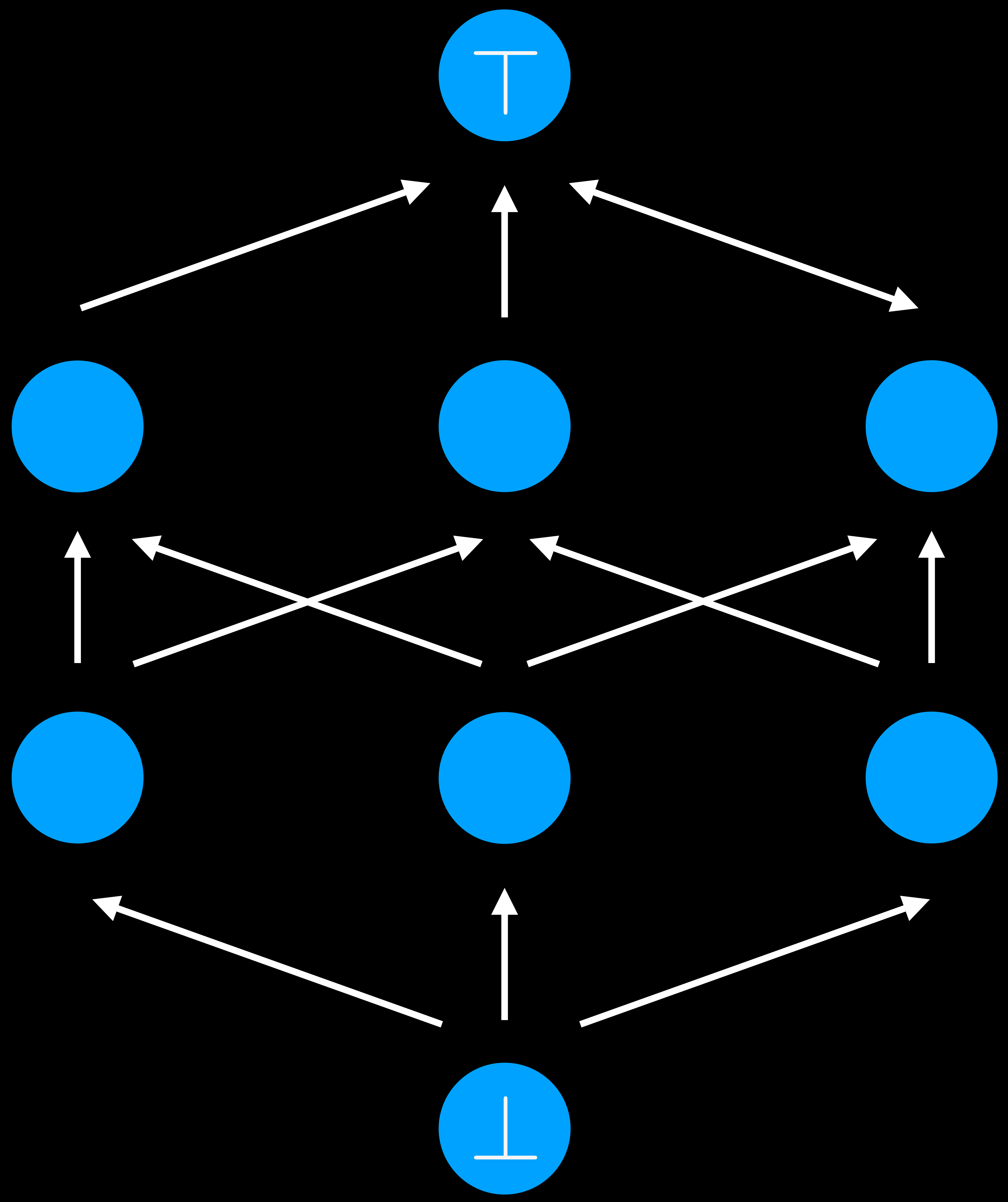
Int Bool Cat Dog Socket Image List AbstractProxyFactorySingletonBean

TCPSocket LinkedList

Nothing

any value is an instance of **Any**

what is an instance of **Nothing**?



lattices

joins and meets

consequences of the bottom type?

ascription and casting

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \text{ as } \tau : \tau}$$

ascription and casting

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \text{ as } \tau : \tau}$$

completely useless, right?

ascription and casting

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e \text{ as } \tau : \tau}$$

ascription and casting

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e \text{ as } \tau : \tau}$$

how do we make this safe at runtime?

functions

$$\frac{\sigma_1 <: \tau_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}$$

functions

$$\frac{\sigma_1 <: \tau_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}$$

functions

$$\frac{\tau_1 <: \sigma_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}$$

functions are ***covariant*** in the ***codomain***
and ***contravariant*** in the ***domain***!

references

$$\frac{\sigma <: \tau}{\sigma \ \mathsf{ref} <: \tau \ \mathsf{ref}}$$

references

$$\frac{\sigma <: \tau}{\sigma\ \mathrm{ref} <: \tau\ \mathrm{ref}}$$

references

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma \text{ ref} <: \tau \text{ ref}}$$

references

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma \text{ ref} <: \tau \text{ ref}}$$

in effect, the types have to be ***equal***,
making references ***invariant***!
(modulo field reordering)

sources and sinks

$$\frac{\Gamma \vdash e : \tau \ \texttt{src}}{\Gamma \vdash\ !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \ \texttt{sink} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \coloneqq e_2 : \texttt{unit}}$$

also called “capabilities”

sources and sinks

$$\frac{\sigma <: \tau}{\sigma\ \mathtt{src} <: \tau\ \mathtt{src}} \qquad \frac{\tau <: \sigma}{\sigma\ \mathtt{sink} <: \tau\ \mathtt{sink}}$$

$$\frac{}{\tau\ \mathtt{ref} <: \tau\ \mathtt{src}} \qquad \frac{}{\tau\ \mathtt{ref} <: \tau\ \mathtt{sink}}$$

arrays

generalized references, so invariant?

arrays

$$\frac{\sigma <: \tau}{\sigma[\,] <: \tau[\,]}$$

what problems does this cause?

logic

$$\frac{\sigma <: \tau_1 \quad \sigma <: \tau_2}{\sigma <: \tau_1 \wedge \tau_2}$$

$$\frac{\sigma <: \tau_1}{\sigma <: \tau_1 \vee \tau_2} \qquad \frac{\sigma <: \tau_2}{\sigma <: \tau_1 \vee \tau_2}$$

```
List<?>    // just ‘a list
```

```
List<? extends Animal>  // no ML equiv.
```

```
List<? extends Any>
```

```
List<? super Cat>
```

```
List<? extends Noisy & Pettable>
```

```
List<? extends Animal & Comparable<?>>
```

“*F*-bounded polymorphism”

bounded quantification is a rich topic…

typechecking easily made undecidable!

```
List<? extends Integer & GreaterThan<1>>
```

parametric vs. inclusion polymorphism

closed vs. open type extension

behavioral subtyping

refinement types

dependent types

type theory becomes richer

but checking becomes undecidable…

and OOP dogma shows up too (cf. Liskov)

further reading

ch. 24, 25 of *PFPL* (by Harper)

ch. 15, 16, 26, 28 of *TaPL* (by Pierce)