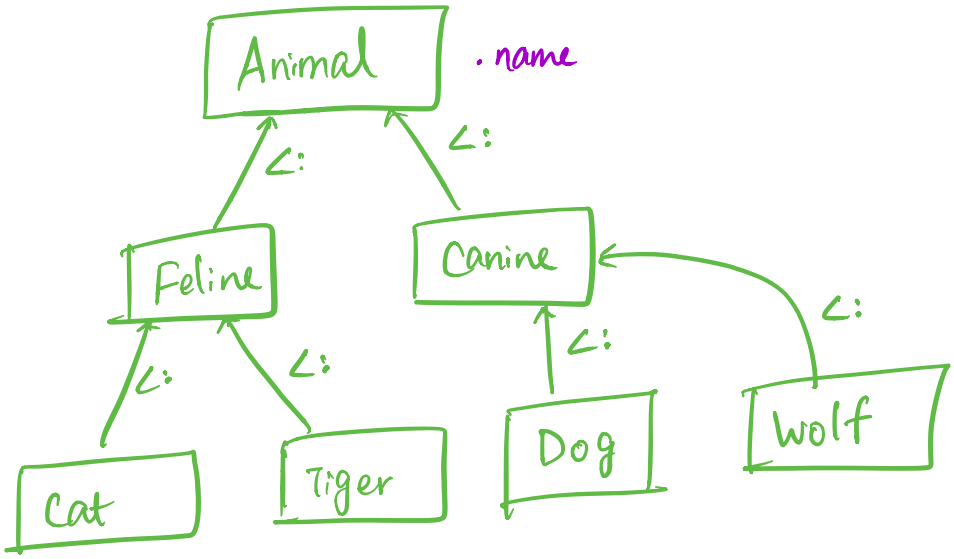
Subtyping



Cat <: Feline <: Animal

In OO, Subclass <: Superclass

In this Problem set,

* Extend STLC w/ tuple type w/
  + depth subtyping
  + width subtyping
* Prove type soundness

"well typed Programs with subtypes do not ~~go wrong~~ get stuck".

Syntax

Expressions $e := x \mid \lambda x.e \mid e_1\ e_2 \mid \emptyset \mid (e_1, e_2) \mid e\#n$

- $\emptyset$: empty tuple expression
- (e_1, e_2): tuple cons
- $e\#n$: Projection

Values $v := \lambda x.e \mid \emptyset \mid (v_1, v_2)$

- $\emptyset$: empty tuple value

Examples

$\emptyset$ is a valid expression

$(\lambda x.x, \emptyset)$ is a valid expression

$(\lambda x.x, \emptyset)\#0$ is a valid expression

$(\lambda x.x, (\lambda y.y, \emptyset))\#1$ is a valid expression

~~$(\lambda x.x, \lambda y.y)$ is invalid. why?~~

↳ must be a tuple expression.

Operational semantics

$(v_1, v_2)\#0 \longrightarrow_0 v_1$ // List.nth

$(v_1, v_2)\#(n+1) \longrightarrow_0 v_2\#n$

Types

Types $\tau := \tau_1 \to \tau_2 \mid \emptyset \mid (\tau_1, \tau_2)$

↳ empty tuple type

$$\frac{}{\Gamma \vdash \emptyset : \emptyset}\ [\text{TupleNil}] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)}\ [\text{Tuple Cons}]$$

- $\emptyset$ (value): Tuple Nil value
- $\emptyset$ (type): Tuple Nil Type

$$\frac{\Gamma \vdash e : \tau' \quad \tau' @ n = \tau}{\Gamma \vdash e \# n : \tau} \; [\text{Proj}]$$

@ (Type-level Projection)

$$(\tau_1, \tau_2) @ 0 = \tau_1$$

$$(\tau_1, \tau_2) @ (n+1) = \tau_2 @ n$$

Example

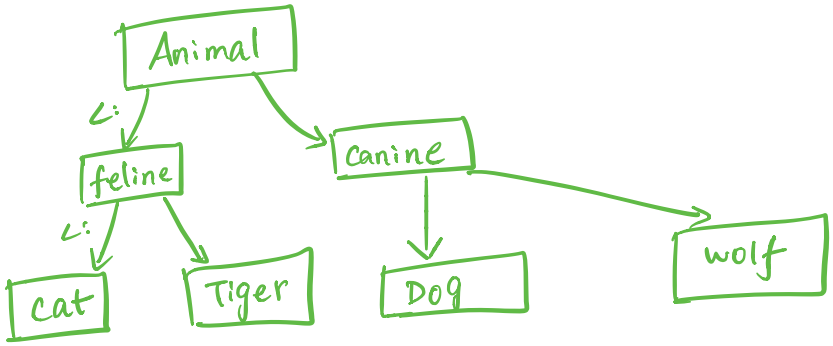
$$\frac{\dfrac{\dfrac{}{y : \emptyset \vdash y : \emptyset}\checkmark \quad \dfrac{}{y : \emptyset \vdash \emptyset : \emptyset}\checkmark}{y : \emptyset \vdash (y, \emptyset) : (\emptyset, \emptyset)}}{\vdash \lambda y.(y, \emptyset) : \emptyset \to (\emptyset, \emptyset)} \quad \frac{}{\vdash \emptyset : \emptyset}\checkmark$$

$$\frac{\dfrac{\dfrac{}{x : \emptyset \vdash x : \emptyset}\checkmark}{\vdash \lambda x.x : \emptyset \to \emptyset} \quad \vdash (\lambda y.(y, \emptyset), \emptyset) : (\emptyset \to (\emptyset, \emptyset), \emptyset)}{\vdash (\lambda x.x, (\lambda y.(y, \emptyset), \emptyset)) : (\emptyset \to \emptyset, (\emptyset \to (\emptyset, \emptyset), \emptyset))} \quad (\emptyset \to \emptyset, (\emptyset \to (\emptyset, \emptyset), \emptyset)) @ 1 = \emptyset \to (\emptyset, \emptyset)$$

$$\vdash (\lambda x.x, (\lambda y.(y, \emptyset), \emptyset)) \#1 : \emptyset \to (\emptyset, \emptyset)$$

Subtyping ($<:$)

* $\tau_1 <: \tau_2$ read as τ_1 subtype of τ_2
* Given an expression e with type τ_1, if $\tau_1 <: \tau_2$, you can consider $e : \tau_2$



Henry : ~~cat~~ feline

⇓

object (value)

retract_claw : feline → unit

↓

accepts cat, tiger but not dog, wolf

$$\frac{}{\phi <: \phi}$$

$$\frac{T_1 <: T_1' \quad T_2 <: T_2'}{(T_1, T_2) <: (T_1', T_2')} \text{ (depth subtyping)}$$

↳ (cat, Tiger) <: (Animal, Feline)

$$\frac{}{(T_1, T_2) <: \phi} \text{ (width subtyping)}$$

Consider the expression:

$f = \lambda t.(t\#0)\ \phi$

extracts the zeroth element of the tuple and applies it to ϕ

we can give this function a type:

$(\phi \to \phi, \phi) \to \phi$

I can apply this function to a tuple of length 1.

$f\ (\lambda x.x, \phi)$

But since the function only will access the zeroth element, I can also Presumably apply this function to:

$f\ (\lambda x.x, (\lambda y.y, \phi))$

doesn't matter

How do we justify this application? we make the type of i.e,

$(\phi \to \phi, (\phi \to \phi, \phi))$

a subtype of

$(\phi \to \phi, \phi)$

(by function subtyping) (* to be seen *)

$\emptyset \to \emptyset <: \emptyset \to \emptyset$

(by width subtyping)

$(\emptyset \to \emptyset, \emptyset) <: \emptyset$

(by depth subtyping)

$(\emptyset \to \emptyset, (\emptyset \to \emptyset, \emptyset)) <: (\emptyset \to \emptyset, \emptyset)$

Subtyping function types

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'}$$

Observe the reversed order!

Consider

```
let retract_claw2 (f: feline -> feline) : unit =
    retract_claw (f tiger1)
```

Assume every animal has a name accessed with .name accessor

```
(* returns a cat with the same name as animal *)
let mk_cat_with_name_of (a: animal) : cat =
    let c = mk_cat () in
    c.name <- a.name;
    c
```

animal -> cat

Can I apply retract_claws2 (mk_cat_with_name_of)?

Should be ok!

retract_claws2 mk_cat_with_name_of

$\rightarrow_\beta$ retract_claw (make_cat_with_name_of tiger1)

$\rightarrow_\beta$ retract_claw cat1

$\rightarrow_\beta$ ()

How can you justify this?

$$\frac{\text{feline} <: \text{animal} \quad \text{cat} <: \text{feline}}{\text{animal} \rightarrow \text{cat} <: \text{feline} \rightarrow \text{feline}}$$

order is reversed.

Intuition: You can substitute a function type with another expect less and offer more