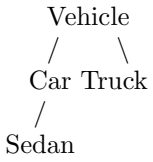
1 Introduction

From the previous lecture, we have the subtyping judgement $\vdash \tau \leq \tau'$ and the subsumption rule

$$\frac{\Gamma \vdash e : \tau \qquad \vdash \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

We can also think of the subtyping judgement as a statement about type inclusion:
$\tau \leq \tau' \Leftrightarrow \mathcal{T}[\![\tau]\!] \subseteq \mathcal{T}[\![\tau']\!]$

Ultimately, our goal is to model object-oriented languages. Specifically, we need some way to model type hierarchies like the following:



In OCaml, we might implement this as

```
type Vehicle : Car of ... \\
   | Truck of ...
```

and use the type `Vehicle` like this:

```
let v: Vehicle = ...
in nWheels =
    case v of Car -> 4
    | Truck -> 18
    ...
```

Adding subtypes will be cumbersome: for each subtype we add, we'll have to manually add a clause to each method of the supertype. Our subtyping model should provide more extensibility.

Another way of thinking about the type inclusion relationship is "τ is inhabited" $\Rightarrow$ "τ' is inhabited", or put another way, $(\phi \Rightarrow \phi')$ where ϕ is the assertion that τ is inhabited and ϕ' the corresponding assertion for τ'. According to the Curry-Howard isomorphism, this leads us to suspect the existence of a term of type $(\tau \rightarrow \tau')$, which we will call a *coercion function* from τ to τ'. This leads to an alternate, simpler way of interpreting the subtype relation on two types that is based on coercion rather than on type inclusion. The coercion function interpretation allows us to interpret types as underlying semantic domains in the same way that we did before subtyping was introduced.

2 Coercion function

We can also define a new operator Θ, which is a coercion from one type to another. For example, $\Theta[\![\vdash \tau \leq \tau']\!]$ is a coercion from type τ to type τ'. Using this, we can replace uses of the subsumption rule in typing proofs. So if we see

$$\frac{\Gamma \vdash e : \tau'' \qquad \vdash \tau'' \leq \tau'}{\Gamma \vdash e : \tau'}$$

in the proof tree of $\Gamma \vdash e_0 : \tau$, we can replace the use of the subsumption rule with a typing of an application of a coercion function:

$$\frac{\Gamma \vdash e' : \tau' \qquad \vdash \Theta[\![\tau' \leq \tau'']\!] : \tau' \to \tau''}{\Gamma \vdash \Theta[\![\tau' \leq \tau'']\!] e' : \tau''}$$

Note that e'_0 is the same as e_0 but with the possible use of coercions inside.

We can define the coercion rules as follows:

$$\begin{aligned}
\Theta[\![\tau \leq \tau]\!] &= \lambda x : \tau . x \\
\Theta\left[\!\!\left[\frac{\tau \leq \tau'' \qquad \tau'' \leq \tau'}{\tau \leq \tau'} \right]\!\!\right] &= \lambda x : \tau . \Theta[\![\tau'' \leq \tau']\!](\Theta[\![\tau \leq \tau'']\!] x) \\
\Theta[\![\tau \leq \#u]\!] &= \lambda x : \tau . \#u \\
\Theta[\![\{l_1 : \tau_1 ... l_m : \tau_m\} \leq \{l_1 : \tau_1 ... l_n : \tau_n\}]\!] &= \lambda x : \{l_1 : \tau_1 ... l_m \tau_m\} . \{l_1 = x.l_1 ... l_n = x.l_n\} \qquad \text{(where } m \geq n\text{)} \\
\Theta[\![\{l_1 : \tau_1 ... l_m : \tau_m\} \leq \{l_1 : \tau'_1 ... l_m : \tau'_m\}]\!] &= \lambda x : \{l_1 : \tau_1 ... l_m \tau_m\} . \{l_1 = \Theta[\![\tau_1 \to \tau'_1]\!] x.l_1 ...\} \\
\Theta\left[\!\!\left[\frac{\tau_1 \leq \tau'_1 \qquad \tau'_2 \leq \tau_2}{\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2} \right]\!\!\right] &= \lambda f : \tau_1 \to \tau_2 . \lambda y : \tau'_1 \Theta[\![\tau_2 \leq \tau'_2]\!](f(\Theta[\![\tau'_1 \leq \tau_1]\!] y))
\end{aligned}$$

3 Objects and Classes

With our typing model complete, we'll now begin to explore objects and classes. Consider the following Java implementation of integer sets:

```
class intset {
        intset union(intset S);
        boolean contains(int);
        int value;
        intset left,right;
}
```

With recursive types and records we can translate this implementation to lambda calculus :

$$\mathsf{intset} = \mu S.\{\mathsf{union} : S \to S, contains : int \to bool, value : int, left, right : S + 1\}$$

A program using this definition would probably include code that looked like the following:

```
let s = fold_intset (rec this: { union : intset → intset, ... } .
        { union = λ s' : intset . ....
        , contains = λ n : int . if n = this.value then #t
            else if m ⟨ this.value
                then case this.left of
                  #u . #f
                  s' . ((unfold s').contains) n
.
.
.
```

But this implementation is inadequate. The internals of the class are fully exposed to any other objects or functions that might use it. We need some way of providing a restricted interface to our objects and classes.

4 Encapsulation/Information Hiding

While we can encode objects currently, we are missing one of the key concepts of Object Oriented programming, which is information hiding. We can encode this with use of existential types, which again map closely to the mathematical equivalent by the Curry-Howard Isomorphism. Basically, $\exists X.\tau \Leftrightarrow \exists X.\phi$.

The idea is that there exists some type X such that we can make something look like τ. The type X then becomes the information that we want to hide. To hide information, we define a new construct: $[\tau, v] : \exists X.\sigma$ where $v : \sigma\{\tau/X\}$. We also introduce two new operators which work on this construct, pack and unpack.

$$\mathsf{pack}_{\exists X.\sigma}[\tau, e] \qquad \text{(introduction form)}$$
$$\mathsf{unpack}\ e_1\ \mathsf{as}\ [Y, x]\ \mathsf{in}\ e_2 \qquad \text{(elimination form)}$$
$$\mathsf{unpack}\ (\mathsf{pack}_{\exists X.\sigma}[\tau, e])\ \mathsf{as}\ [Y, x]\ \mathsf{in}\ e_2 \longmapsto e_2\{\tau/Y, v/x\}$$

unpack is similar to a let expression where τ is bound to Y and e is bound to x. In this way, the actual type is hidden within τ. Then in the unpack expression Y provides the interface to the hidden type which won't affect evaluation.

Evaluation contexts are extended in the obvious way for pack and unpack:

$$C ::= \ldots \mid \mathsf{pack}_{\exists X.\sigma}[\tau, C] \mid \mathsf{unpack}\ C\ \mathsf{as}\ [Y, x]\ \mathsf{in}\ e_2$$

In the following simple integer object example, the integer type is packed into X; the code in the unpack expression is able to operate on the integer and the int→bool function, without knowing they are of those types. We need only think of (π_1 x) as having type Y. p has the existential type ∃X.X * (X→bool):

$\mathsf{let}\ \mathsf{p} = \mathsf{pack}_{\exists X.X*(X\to bool)}[\mathsf{int}, \langle 5, \lambda \mathsf{z}{:}\mathsf{int}.\,(\mathsf{z} = 4)\rangle]\ \mathsf{in}$
$\quad \mathsf{unpack}\ \mathsf{p}\ \mathsf{as}\ [T, \mathsf{x}]\ \mathsf{in}\ (\pi_2\ \mathsf{x})\ (\pi_1\ \mathsf{x})$