1. The Syntax of Types

Here we show the syntax of types in Wyvern:

$$\begin{array}{rcl} \tau & ::= & \tau \to \tau \mid \forall t.\tau \\ & \mid & \mathbf{rec}\ t.\tau \mid t \\ & \mid & \{\overline{\delta}\} \mid p.t \\ \\ \delta & ::= & \mathbf{val}\ f : \tau \\ & \mid & \mathbf{type}\ t = \tau \\ & \mid & \mathbf{type}\ t \\ \\ p & ::= & x \mid p.f \end{array} \tag{1}$$

Type application $\tau[t = \tau']$ is not relevant to the theory of subtyping because it can be reduced away syntactically to $[\tau'/t]\tau$ before the subtyping test.

TODO: add intersection (see example below). This may subsume $\tau[t = \tau']$ anyway.

This type theory does not provide bounded quantification. The rationale is that most of the value in polymorphism comes when the target type is treated completely polymorphically, and that integrating polymorphism and subtyping comes at a relatively high cost. Wyvern remains quite expressive. For example, here is an example showing support for sorted lists that accept `Comparable` elements, where in other languages `Comparable` would be a type class or a F-bounded type:

```
type ComparableClass
    type InstanceType
    def compare(e1:InstanceType,e2:InstanceType):boolean

type Comparable
    type Self
    def comparableClass():ComparableClass[Self]

class SortedList
    class def make[E](c:ComparableClass[E]):SortedList[E]
        makeWithCompare[E](c.compare)
```

```
class def makeSingleton[E](e:E & Comparable[E]):SortedList[E]
    val l = makeWithCompare[E](e.comparableClass().compare)
    l.add(e)
    l

class def makeWithCompare[E](compare:(E,E) -> boolean):SortedList[E] = ...

// all the other list stuff goes here ...
```

Compare this to a system with F-bounded polymorphism. In the code above we have to put compare() in the class, but this is arguably sensible anyway since compare is a binary method (otherwise we would not be using a type class like structure). There is a little boilerplate linking the class and instance, which perhaps could in typical cases be provided with syntactic sugar. Finally, we have to write `E & Comparable[E]` in one place, but this is arguably easier to understand than the F-bounded solution `E extends Comparable[E]`.

And here are the subtyping rules:

$$\boxed{\Gamma \vdash \tau_1 <: \tau_2}$$

$$\frac{}{\Gamma \vdash \tau = \tau}\ \textsc{SubReflex}$$

$$\frac{\Gamma \vdash \tau_2 <: \tau_2' \quad \Gamma \vdash \tau_1' <: \tau_1}{\Gamma \vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}\ \textsc{SubArrow}$$

$$\frac{\Gamma, t\ \mathbf{type} \vdash \tau_1 \simeq \tau_2}{\Gamma \vdash \forall t.\tau_1 <: \forall t.\tau_2}\ \textsc{SubForall}$$

$$\frac{\Gamma, t <: t' \vdash \tau <: \tau'}{\Gamma \vdash \mathbf{rec}\ t.\tau <: \mathbf{rec}\ t'.\tau'}\ \textsc{SubRec}$$

$$\frac{}{\Gamma, \tau_1 <: \tau_2 \vdash \tau_1 <: \tau_2}\ \textsc{SubCtx}$$

$$\frac{\Gamma \vdash \{\overline{\delta}\} <: \{\overline{\delta}'\}}{\Gamma \vdash \{\overline{\delta}, \overline{\delta}_2\} <: \{\overline{\delta}'\}}\ \textsc{SubWidth}$$

$$\frac{\Gamma \vdash \overline{\delta} <: \overline{\delta}'}{\Gamma \vdash \{\overline{\delta}\} <: \{\overline{\delta}'\}}\ \textsc{SubDepth}$$

$$\frac{\Gamma \vdash \tau_1 \simeq \tau_1' \quad \Gamma \vdash \tau_2 \simeq \tau_2' \quad \Gamma \vdash \tau_1' <: \tau_2'}{\Gamma \vdash \tau_1 <: \tau_2}\ \textsc{SubUnalias} \tag{2}$$

$$\boxed{\Gamma \vdash \delta_1 <: \delta_2}$$

$$\frac{}{\Gamma \vdash \delta = \delta}\ \textsc{SubMemberReflex}$$

$$\frac{\Gamma \vdash \tau <: \tau'}{\Gamma \vdash \mathbf{val}\ f : \tau <: \mathbf{val}\ f : \tau'}\ \textsc{SubField}$$

$$\frac{}{\Gamma \vdash \mathbf{type}\ t = \tau <: \mathbf{type}\ t}\ \textsc{SubAlias}$$

$$\boxed{\Gamma \vdash \tau_1 \simeq \tau_2}$$

$$\frac{\Gamma \vdash p : \tau_p \quad \tau_p[t] = \tau_t \quad \Gamma \vdash \tau_t \simeq \tau}{\Gamma \vdash p.t \simeq \tau}\ \textsc{TypeExpand}$$