1 Interpreting subtyping as coercion

We can interpret the subtyping relation on types in a natural manner as a subtype relation on the interpretation of the types as sets or domains:

$$\tau_1 \leq \tau_2 \qquad \Longleftrightarrow \qquad \mathcal{T}(\tau_1) \subseteq \mathcal{T}(\tau_2)$$

According to the Curry-Howard isomorphism, we interpret a type as an assertion that the type is inhabited: that there exists a (non-bottom) value of that type. The subset relationship between $\mathcal{T}(\tau_1)$ and $\mathcal{T}(\tau_2)$ means that τ_2 is inhabited if τ_1 is, or $\tau_1 \Rightarrow \tau_2$. Applying the Curry-Howard isomorphism, this formula implies the existence of a function of type $\tau_1 \to \tau_2$. To make this more concrete, we can introduce a function Θ that verifies the subtyping relation instance by producing such a value:

$$\tau_1 \leq \tau_2 \qquad \Longrightarrow \qquad \Theta(\tau_1 \leq \tau_2) : \tau_1 \to \tau_2$$

So that the subsumption rule

$$\frac{\Gamma \vdash e : \tau_1 \quad \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}$$

becomes

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \Theta(\tau_1 \leq \tau_2) : \tau_1 \to \tau_2}{\Gamma \vdash (\Theta(\tau_1 \leq \tau_2) \quad e) : \tau_2}$$

We still have to define the coercion function:

$$\Theta(\tau \leq \tau) = \lambda x : \tau . x$$

$$\Theta(\tau \leq 1) = \lambda x : \tau . \#\mathsf{u}$$

For record types, we can associate separate coercion functions with each of the width and depth subtyping rules:

$$\Theta(\{l_1 : \tau_1, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau_1, \ldots, l_m : \tau_m\}) = \lambda x.\{l_1 = x.l_1, \ldots, l_m = x.l_m\} \qquad (m \leq n)$$

$$\Theta(\{l_1 : \tau_1, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau'_1, \ldots, l_n : \tau'_n\}) = \lambda x.\{l_1 = (\Theta(\tau_1 \leq \tau'_1) \quad x.l_1), \ldots, l_n = (\Theta(\tau_n \leq \tau'_n) \quad x.l_n)\}$$

$$\Theta(\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2) = \lambda f : (\tau_1 \to \tau_2).(\Theta(\tau_2 \leq \tau'_2) \quad f \quad (\Theta(\tau'_1 \leq \tau_1) \quad x))$$

Note that the coercion function is defined by induction on the derivation of the relation $\tau_1 \leq \tau_2$ – in order to define the value of $\Theta(\tau_1 \leq \tau_2)$ one has to follow the derivation of the subtyping relation instance $\vdash \tau_1 \leq \tau_2$, and define Θ on each instance in the proof tree.

2 Typed translation

We have seen in the past definitions of translation functions – functions which transform expressions in one given language to equivalent expressions in another. Translation of typed languages achieves even more: it translates derivations of type judgements, so that no rejudging has to be made in the new language. Such a translation function will have the form:

$$\mathcal{D}[\![\Gamma \vdash e : \tau]\!] = \Gamma \vdash e' : \tau$$

Now we present some of the rules which define the translation function $\mathcal{D}$. For derivation steps using the subsumption rule:

$$\frac{\mathcal{D}[\![\Gamma \vdash e : \tau_1]\!] = \Gamma \vdash e' : \tau_1 \quad \Gamma \vdash \Theta(\tau_1 \leq \tau_2) : \tau_1 \to \tau_2}{\mathcal{D}[\![\Gamma \vdash e : \tau_2]\!] = \Gamma \vdash (\Theta(\tau_1 \leq \tau_2) \quad e') : \tau_2}$$

More rules:

$$\mathcal{D}[\![\Gamma, x:\tau \vdash x:\tau]\!] = \Gamma, x:\tau \vdash x:\tau$$

$$\frac{\mathcal{D}[\![\Gamma \vdash e_1 : \tau \to \tau']\!] = \Gamma \vdash e_1' : \tau \to \tau' \quad \mathcal{D}[\![\Gamma \vdash e_2 : \tau]\!] = \Gamma \vdash e_2' : \tau}{\mathcal{D}[\![\Gamma \vdash (e_1 \quad e_2) : \tau']\!] = \Gamma \vdash (e_1' \quad e_2') : \tau'}$$

$$\mathcal{D}[\![\Gamma \vdash (\lambda x:\tau.e) : \tau \to \tau']\!] = \Gamma \vdash (\lambda x:\tau.e') : \tau \to \tau'$$

where $\mathcal{D}[\![\Gamma, x:\tau \vdash e:\tau']\!] = \Gamma, x:\tau \vdash e' : \tau'$.

3 Abstract Data Types (ADTs)

In our programs we would like to be able to limit the interaction between pieces of code. We should able to provide an interface by which seperate parts can interact, and force that to be the only method of interaction. In Java/C++ this is achieved using public/private fields in classes. Public fields can be accessed by anyone, while private fields can only be referred to within their containing class.

Here is a Java-like example, implementing an integer set as a binary tree:

```
class intset{
    public boolean contains(int x);
    public intset union(intset i);
    private intset left, right;
    private int value;
}
```

The public interface has only the operations union and contains. The binary structure should be transparent from the outside.

We will hide things by using Existential Data Types

4 Existential Data Types

The Curry-Howard isomorphism gives us $\exists X.\tau \leftrightarrow \exists x.\phi$. The existential type should be in some sense dual to the universal. We will think of $\exists X.\tau$ as a pair $[\tau', v]$ with $v : \tau\{\tau'/X\}$. Here τ' is the witness type. We contstruct v and hide τ' using $\exists X$. Externally we can't get τ' back.

Here are the rules for the isomorphism:

$$\frac{\Delta;\Gamma \vdash \phi\{\phi'/x\}}{\Delta;\Gamma \vdash \exists x.\phi} \qquad \frac{\Delta;\Gamma \vdash \exists x.\phi_1 \qquad \Delta, x';\Gamma,\phi_1\{x'/x\} \vdash \phi_2}{\Delta;\Gamma \vdash \phi_2} \quad \text{with } x' \text{ not free in } \Gamma, \phi_1, \phi_2 \text{ (“fresh”)}$$

And the corresponding type rules:

$$\frac{\Delta;\Gamma \vdash e : \sigma\{\tau/X\}}{\Delta;\Gamma \vdash \mathsf{pack}[x=\tau, e] : \exists X.\tau} \qquad \frac{\Delta;\Gamma \vdash e_1 : \exists X, \sigma_1 \qquad \Delta, X';\Gamma, X:\sigma_1\{X'/X\} \vdash e_2 : \sigma_2}{\Delta;\Gamma \vdash \mathsf{unpack}\ e_1\ \text{as}\ [\mathrm{X}', \mathrm{X}]\ \text{in}\ \mathrm{e}_2 : \sigma_2} \quad \text{with } X' \text{ “fresh”}$$