CS611 Lecture 35: Subtype polymorphism — November 19, 1999

Rama, Nystrom

1 More on the Curry-Howard Isomorphism

Recall that the Curry-Howard isomorphism gives us a correspondence between a polymorphic types and formulae in constructive logic. For example,

$$\alpha * \beta \approx \alpha \wedge \beta.$$

Since in constructive logic,

$$\alpha \wedge \beta \approx \forall\gamma.(\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \gamma,$$

we can map back to types to get

$$\alpha * \beta \approx \forall\gamma.(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \gamma.$$

We can therefore construct a function from a value of type $\alpha * \beta$ to a value of type $\forall\gamma.(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \gamma$. We'll use $[\![\alpha * \beta]\!]$ to denote the type $\forall\gamma.(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \gamma$. Such a function looks like

$$\lambda p : [\![\alpha * \beta]\!].\, \langle(\mathsf{pi}_1[\alpha, \beta]\ p), (\mathsf{pi}_2[\alpha, \beta]\ p)\rangle$$

The inverse function is

$$\lambda p : \alpha * \beta.\, (\mathsf{pair}[\alpha, \beta]\ \langle\ ()\ \pi_1)p, (\pi_2\ p)\rangle$$

Now, one might think that we could apply a similar transformation to sum types using the correspondence with logical or:

$$\begin{aligned} \alpha \vee \beta &= \neg\alpha \vee \beta \\ &= (\forall\gamma.\alpha \Rightarrow \gamma) \Rightarrow \beta \end{aligned}$$

This correspondence holds in classical logic and suggest that we should be able to model a sum type $\alpha + \beta$ as a value of the type $(\forall\gamma.\alpha \rightarrow \gamma) \rightarrow \beta$. But, remember we are working in constructive (aka *intuitionistic*) logic, not classical first-order logic. In constructive logic, the logical implication above holds in one direction only, and there is therefore no isomorphism between the two types. However, there is another way to desugar sum types into the polymorphic lambda calculus using the Curry-Howard isomorphism, but this is left as a homework exercise.

2 Subtype Polymorphism

Using the polymorphic lambda-calculus, we can construct a data type for a list of elements of type α: α list. However, if we want a list that contains many different element types, we will have to use a sum type.

$$\alpha = (\mathsf{rhino} + \mathsf{elephant} + \mathsf{bear})$$

But, what if we want to add alligator to our list of large scary animals. We will have to redefine α. We would like something more extensible.

2.1 Subtyping

Simula 67 introduced the notion of subtyping. Thus we can create an animal list, where

$$\begin{aligned} \mathsf{rhino} &\leq \mathsf{animal} \\ \mathsf{elephant} &\leq \mathsf{animal} \\ \mathsf{bear} &\leq \mathsf{animal} \\ \mathsf{alligator} &\leq \mathsf{animal} \end{aligned}$$

When we create a new animal type, we can make it a subtype of animal and leave animal list unchanged.

We use $T_1 \leq T_2$ to mean every value of T_1 is also a value of T_2. In other words, $\mathcal{T}[\![T_1]\!] \subseteq \mathcal{T}[\![T_2]\!]$. Since $v \in \mathcal{T}[\![T_1]\!] \Rightarrow v \in \mathcal{T}[\![T_2]\!]$, there is a coersion function $\theta(T_1 \leq T_2)$ of type $T_1 \to T_2$.

This leads to a new type judgment.

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

This rule is called the *subsumption rule*.

2.2 Subtyping as an ordering relation

$\leq$ is an ordering relation on type expressions τ. We might ask what properties hold on the relation. First, reflexivity holds:

$$\frac{}{\tau \leq \tau}$$

Transitivity also holds:

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

The coercion function is simply the composition:

$$\theta(\tau_1 \leq \tau_3) = \theta(\tau_2 \leq \tau_3) \circ \theta(\tau_1 \leq \tau_2)$$

Together these two properties define a pre-order on type expressions.

Anti-symmetry holds in some languages but not others. Anti-symmetry would require that if two types are subtypes of each other, they are syntactically identical.

$$\tau_1 \equiv \tau_2 \Leftrightarrow \tau_1 \leq \tau_2 \wedge \tau_2 \leq \tau_1$$

However, in many languages two types can have a mutual subtype relation without being syntactically identical, or even without representing the same set of values. For example, in Modula-3, array of char and packed array of char can be used interchangeably. Thus,

$$\textsf{array of char} \leq \textsf{packed array of char}$$

and

$$\textsf{packed array of char} \leq \textsf{array of char}.$$

The two types are equivalent but not equal to each other.

We have already seen an example of type equivalence when we studied recursive types: $\mu\alpha.\tau \cong \tau[\mu\alpha.\tau]$. For example, in Modula-3, the following two types are equivalent.

$$\begin{array}{lcl}
T = \textsf{object} & \cong & T' = \textsf{object} \\
\quad x_1 : \tau_1 & & \quad x_1 : \tau_1 \\
\quad x_2 : T & & \quad x_2 : \textsf{object} \\
\textsf{end} & & \qquad x_1 : \tau_1 \\
& & \qquad x_2 : T' \\
& & \quad \textsf{end} \\
& & \textsf{end}
\end{array}$$

Modula-3 has structural type equivalence. In many languages, such as C, the name of a structure or object type would distinguish the types.

2.3 Subtyping of derived types

2.3.1 Cartesian Product

The subtype rule and coersion function for cartesian product of two types is very easy to define.

$$\frac{\tau_1 \leq \tau_1' \,,\, \tau_2 \leq \tau_2'}{\tau_1 * \tau_2 \leq \tau_1' * \tau_2'}$$

$$\theta(\tau_1 * \tau_2 \leq \tau_1' * \tau_2') = \lambda p : \tau_1 * \tau_2 \langle \theta(\tau_1 \leq \tau_1')\Pi_1 p \,,\, \theta(\tau_2 \leq \tau_2')\Pi_2 p \rangle$$

2.3.2 Sum Type

The subtype rule and coersion function for sum of two types is defined similar to cartesian products.

2.3.3 Function Type

Let us consider this case by case. Consider two types $\tau_1 \to \tau_2$ and $\tau_1 \to \tau_2'$. If we want the former to be a subtype of later, then we should be able to use an expression of the later type wherever an expression of former type appears without violating the domain equations. This is true if $\tau_2 \leq \tau_2'$. The subtyping is said to be monotonic or covariant in results. Now consider the two types $\tau_1 \to \tau_2$ and $\tau_1' \to \tau_2$. Applying the same logic that an expression of the former type can be replaced by an expression of later type without violating the domain equations, we see that $\tau_2' \leq \tau_2$. Subtyping is said to be anti-monotonic or contravariant in arguments. Thus the subtype inference rule and the coersion function are as defined.

$$\frac{\tau_1' \leq \tau_1 \;,\; \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

$$\theta(\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2') = \lambda f : \tau_1 \to \tau_2 . \lambda x : \tau_1' . \theta(\tau_2 \leq \tau_2')(f(\theta(\tau_1' \leq \tau_1)x))$$

2.3.4 Ref Type

Earlier we had introduced constructs such as (ref e), (e_1 := e_2) and (^e). By creating a new type called ref(τ), we can define the typing rules for these constructs as follows.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash ref\ e : ref(\tau)}$$

$$\frac{\Gamma \vdash e : ref(\tau)}{\Gamma \vdash^{\wedge} e : \tau}$$

$$\frac{\Gamma \vdash e_1 : ref(\tau) \;,\; \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \tau}$$

When is $ref(\tau_1) \leq ref(\tau_2)$? This question is not easy to answer see the following code fragments, for example

x: ref (τ_1) (let (y :ref (τ_2) x)) y := v
x: ref (τ_1) (let (y :ref (τ_2) x)) ^y := v

In the first code, the value v could be of type τ_2 then we would be assigning a value τ_2 to x which is pointing to a value of type τ_1. Hence $\tau_2 \leq \tau_1$. In the second code, v could be of type τ_2 which would require $\tau_1 \leq \tau_2$. Thus ref is neither monotonic nor anti-monotonic. It is said to be invariant.

$$\frac{\tau_1 \cong \tau_2}{ref(\tau_1) \leq ref(\tau 2)}$$

2.3.5 Special type for True and False

Using subtype polymorphism, we can construct type definitions for true and false (the 0 and 1 types). In logic, for all formulas A we have $A \Rightarrow 1$ and $0 \Rightarrow A$, so we would expect that the corresponding subtype relationships could hold:

$$\frac{}{0 \leq \tau} \;, \forall \tau$$

$$\frac{}{\tau \leq 1} \;, \forall \tau$$

The coercion function for true would look like

$$\theta(\tau \leq 1) = \lambda x : \tau . \# u$$

. A coercion function for false is not really needed because it can never be invoked. *One possible coercion function for false looks like*

$$\theta(0 \leq \tau) = \lambda x : 0.\bot$$

where $\bot$ is a divergent expression. This coercion function relies on the element $\bot$ being a member of every type, which will be the case in a language with the possibility of divergence. Another possibility is to interpret the expression $\bot$ here as one whose evaluation immediately terminates the program.

2.3.6 Special type for continuation

Using subtype polymorphism, the continuation can be defined to have the following type $\tau \to 0$. Where 0 is defined using subtype polymorphism as in previous subsection.