

# Higher-Order Subtyping

Karl Schnaitter

June 8, 2005

## 1 Introduction

In chapter 31 of the textbook, two features are combined. The new language combines bounded quantification with type operators. The motivation to do this is simple: if we have two features that are good, why not combine them? Bounded quantification relies on a subtyping relation, so subtyping must be defined in the new language. We have a subtyping relation for types of kind  $*$ , but we have not yet defined how to identify subtyping relationships for types of other kinds, like  $* \Rightarrow *$ . The goal of chapter 32 is to extend subtyping to be defined on all kinds. This allows bounded quantification and type operators to coexist.

## 2 Design

When bounded quantification was first introduced, there were two rules for determining whether or not  $\forall S_1.S_2 <: \forall T_1.T_2$ . There was a tradeoff between the expressiveness of the “full” rule and the simplicity of the “kernel” rule. When defining higher-order subtyping, we again need to make design decisions based on this tradeoff.

Pierce suggests that the most influential decision is in the design of type operators. When bounded quantification was introduced, we added bounds to type variables. We could consider doing the same thing for type operators. Particularly, we could introduce bounds for the types of arguments to type operators.

Instead, the decision is to keep the same syntax for type operators, which are written in the form  $\lambda X :: K.T$ . This decision is analogous to the design of the simply-typed lambda-calculus. In the abstraction  $\lambda x : T.t$ , we specify that the permissible *values* of  $x$  belong in a particular *type*  $T$ . We cannot make arbitrary restrictions on  $x$ . Now for a type operator in the form  $\lambda X :: K.T$ , we specify that the type variable  $X$  may only assume *types* of the particular *kind*  $K$ . Pierce mentions the option of making more fine-grained restrictions on arguments in type operators. This would introduce syntax in the form  $\lambda X <: T.T$ , and the semantics of the resulting language would be much more complex to deal with.

System  $F_\omega$  and system  $F_{<}$  differ in their design of type-variable bindings. A binding of a context in system  $F_{<}$  is in the form  $X <: T$ , while system  $F_\omega$  uses bindings in the form  $X :: K$ . We can express the latter form in the former form by creating some new notation. We need to express that  $X$  is a subtype of all types of kind  $K$ . Each kind has a maximal type, called  $\text{Top}[K]$ . This type is defined as follows:

$$\begin{aligned} \text{Top}[*] &\stackrel{\text{def}}{=} \text{Top} \\ \text{Top}[K_1 \Rightarrow K_2] &\stackrel{\text{def}}{=} \lambda X :: K_1. \text{Top}[K_2] \end{aligned}$$

With this notation, we can express  $X :: K$  as the subtype relationship  $X <: \text{Top}[K]$ .

In the textbook's specification of the language  $F_{<}^\omega$ , there are many changes to previous languages. Most of these changes are old ideas or reformulations of old ideas. The new ideas are found in the two rules for subtyping that involve type operators.

### Type Operator Abstraction

A type operator specifies the exact kind for its argument. Hence, in order for one operator to “maquerade” as another, it is necessary for the kinds of their arguments to be the same. The other requirement is that after applied, the resulting types should have a covariant relationship. This leads to the following rule:

$$\frac{\Gamma, X <: \text{Top}[K] \vdash S <: T}{\Gamma \vdash \lambda X :: K.S <: \lambda X :: K.T} \text{ (S-ABS)}$$

### Type Operator Application

As we just saw, if type operators are subtypes of one another, their results will have the same subtype relationship. In the antecedent of the previous rule, we require that  $S <: T$  with  $X$  in the context as some abstract (but fixed) type. This suggests the following rule:

$$\frac{\Gamma \vdash S <: T}{\Gamma \vdash S U <: T U} \text{ (S-APP)}$$

It's pretty clear that this rule is sound. Pierce adds that there are more flexible rules that allow the arguments to the operators to be different. It gets very complex because some operators are covariant, and others are contravariant.

## 3 Properties

This language has the standard properties: progress, preservation, minimal types, and difficult metatheory. In class, we made another observation about the subtyping relationship: if  $\Gamma \vdash T :: K$  and  $\Gamma \vdash S <: T$ , then  $\Gamma \vdash S :: K$ . In other words, types of different kinds are incomparable. This can be proven easily by induction on the derivation of  $S <: T$ . This also ensures that the reformulation of  $X :: K$  as  $X <: \text{Top}[K]$  is sound.

In the proof, the cases for type operators are slightly interesting. If the last rule in the derivation of  $S <: T$  is S-ABS, then the last rule has the form

$$\frac{\Gamma, X <: \text{Top}[K_1] \vdash S_2 <: T_2}{\Gamma \vdash \lambda X :: K_1.S_2 <: \lambda X :: K_1.T_2}$$

Now by induction,  $S_2$  and  $T_2$  have the same kind, say  $K_2$ . Then by the K-ABS rule, we have  $\Gamma \vdash \lambda(X :: K_1.S_2) :: K_1 \Rightarrow K_2$ , and  $\Gamma \vdash \lambda(X :: K_1.T_2) :: K_1 \Rightarrow K_2$ . For the other case, suppose the last rule in the derivation of  $S <: T$  is S-APP. We can write the last rule as

$$\frac{\Gamma \vdash S_1 <: T_1}{\Gamma \vdash S_1 U <: T_1 U} \text{ (S-APP)}$$

Since  $S_1$  and  $T_1$  are being applied, they must be abstraction kinds. By induction,  $S_1$  and  $T_1$  must also be the same kind, say  $K_{11} \Rightarrow K_{12}$ . By the K-APP rule, we have  $\Gamma \vdash S_1 U :: K_{12}$  and  $\Gamma \vdash T_1 U :: K_{12}$ . The remaining cases are even easier.