Subtyping F-bounded types

Dinesh Katiyar

Stanford University
California, USA

katiyar@theory.stanford.edu

(Work in progress with John Mitchell and Sriram Sankar)

[Position paper for the ANSA workshop on F-bounded Quantification, Cambridge, Sept 1992]

Abstract

We look at the subtyping problem for polymorphic types with F-bounded quantification *i.e.* polymorphic types where a bound can be imposed on the type argument (by the subtype relation), and the bound is permitted to refer to the variable itself. We look at different possibilities for the rule and compare them.

Introduction

Bounded quantification is a notion that combines the ideas of subtyping and polymorphism. It was first introduced by Cardelli and Wegner [CW85] in their language FUN. With bounded quantification, one can express a term that is polymorphic over all the subtypes of a given type. We use the notation "$\Lambda t <: T.e$" for such a term. The "bound" T, in essence, specifies the assumptions that have been made about the type t in the code e. We write the type of this term as "$\forall t <: T.T_e$" where T_e is the type of the expression e.

Canning *etal* [CCH+89] have shown that in the bounded polymorphic type $\forall t <: F.T$, it is useful to be able to allow F to refer to the type variable t. This has come to be known as F-bounded quantification. This idea proves very useful when one has subtyping and recursive type definitions, as in most object-oriented languages. To visualize the use of this notion, consider a polymorphic `sort` function, which accepts a list of elements of some type t and uses some fancy sorting algorithm to sort the list. This algorithm however needs to assume the presence of an ordering on the elements of type t. One property of all such types t could be that they all have a `compare` method with type $t \times t \rightarrow \texttt{Bool}$. Now one could imagine writing the code for such a function which takes as an argument a type t which is a subtype of the type that stipulates the presence of a `compare` method. Assuming something like record types, such a type could be written as $\{\texttt{compare}: t \times t \rightarrow \texttt{Bool}\}$. Hence we could give this function the type

$$\forall t <: \{\texttt{compare}: t \times t \rightarrow \texttt{Bool}\}.\ \texttt{list}(t) \rightarrow \texttt{list}(t)$$

We use the notation "$\forall t <: F(t).T$" for polymorphic types with recursive bounds, and terms of such a type are written as "$\Lambda t <: F(t).e$".

The typing rule for the application of an instance of such a type now simply has to check that any type to which such a type abstraction is applied, satisfies the constraint imposed by the recursive bound.

$$(\mathit{poly\_application}) \qquad \frac{\Gamma \vdash e : \forall t{<:}F(t).T \qquad \vdash S <: F(S)}{\Gamma \vdash eS : [S/t]T}$$

Rules for subtyping

The subtyping rule for such types requires some thought. We first take a look at the rule for ordinary bounded quantification. The subtyping rule, which is fairly standard by now, is as follows:

$$(\mathit{poly\_sub}) \qquad \frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma, t{<:}T_2 \vdash T_1' <: T_2'}{\Gamma \vdash \forall t{<:}T_1.T_1' <: \forall t{<:}T_2.T_2'}$$

(As an aside, decidability problems concerning subtype checking, and hence type checking, raised by the use of this rule have been the focus of a fair amount of research lately [Ghe92, Pie92, KS92].)

Given the above rule for the ordinary case, a first guess for the rule with recursive bounds might be

$$(\mathit{f\_poly\_sub_1}) \qquad \frac{\begin{array}{c}\Gamma \vdash G(t) <: F(t) \\ \Gamma \cup \{t{<:}G(t)\} \vdash T_1 <: T_2\end{array}}{\Gamma \vdash \forall t{<:}F(t).T_1 <: \forall t{<:}G(t).T_2}$$

However, the first premise for this rule is clearly too strong, since it requires that $G(t) <: F(t)$ for all types t, while we know that we need this to hold only when $t <: G(t)$. This can be demonstrated by the following subtyping problem:

$$\Gamma \vdash \forall t{<:}\alpha{\rightarrow}(\alpha{\rightarrow}t).T <: \forall t{<:}\alpha{\rightarrow}t.T$$

The subproblem arising from the first premise (the second is trivially true), that is,

$$\Gamma \vdash \alpha{\rightarrow}t <: \alpha{\rightarrow}(\alpha{\rightarrow}t)$$

is provable only if we add to Γ the assumption that $t <: \alpha{\rightarrow}t$.

Hence we modify the subtyping rule to

$$(\mathit{f\_poly\_sub_2}) \qquad \frac{\begin{array}{c}\Gamma \cup \{t{<:}G(t)\} \vdash G(t) <: F(t) \\ \Gamma \cup \{t{<:}G(t)\} \vdash T_1 <: T_2\end{array}}{\Gamma \vdash \forall t{<:}F(t).T_1 <: \forall t{<:}G(t).T_2}$$

However, if one now attempts to deduce the subtyping rule from first principles, the results are a bit different. Suppose some context expects an instance of the type $\forall t{<:}G(t).T_2$, which means it needs a function that takes an argument type satisfying the constraint $t{<:}G(t)$, and returns a value of type T_2. If we instead substitute an instance of the type $\forall t{<:}F(t).T_1$, we are supplying a function that takes a type satisfying $t{<:}F(t)$ and returns a value of type T_1. Now such a substitution will not cause any type errors if

- any type argument that was originally plausible (*i.e.* any type t such that $t{<:}G(t)$) satisfies the constraint of the substituted instance (*i.e.* $t{<:}F(t)$), and
- for any plausible type argument (*i.e.* any type t such that $t{<:}G(t)$), the value returned by the substituted type (*i.e.* instances of T_1) are values that were originally acceptable (*i.e.* instances of T_2).

These requirements lead us to the rule

$$(f\_poly\_sub_3) \qquad \frac{\begin{array}{c}\Gamma \cup \{t{<:}G(t)\} \vdash t <: F(t) \\ \Gamma \cup \{t{<:}G(t)\} \vdash T_1 <: T_2\end{array}}{\Gamma \vdash \forall t{<:}F(t).T_1 <: \forall t{<:}G(t).T_2}$$

Now what exactly is the difference between these last two rules? *f_poly_sub$_3$*, it would seem, is more flexible since it would allow us to deduce (correctly) something like

$$\Phi \vdash \forall t{<:}t.T <: \forall t{<:}\alpha.T$$

since the first premise reduces to

$$t <: \alpha \vdash t <: t$$

which would hold by reflexivity of the subtype relation. If we use *f_poly_sub$_2$*, then this premise reduces to

$$t <: \alpha \vdash \alpha <: t$$

which cannot be shown.

However, one can convincingly argue that in relatively simple type systems, this extra flexibility arises only when one has types of the form $\forall t{<:}t.T$, which seems like a rather awkward formulation of the type $\forall t.T$ since the constraint is trivially satisfied by all types. The argument is based on the fairly standard subtyping rules proposed by Curien and Ghelli for the $F_{<:}$ calculus [CG91], which is a simple typed λ-calculus with bounded quantification. Consider the first premise of *f_poly_sub$_3$*, namely

$$\Gamma \cup \{t{<:}G(t)\} \vdash t <: F(t)$$

In the Curien-Ghelli formulation of the subtype checking algorithm, there are only three ways of proving a subtyping judgement with a type variable on the left hand side:

- using the reflexivity rule, which can be done only if $F(t) = t$. This means the subtype is one of the "awkward" kinds just mentioned above.
- using the rule which says that every type is a subtype of the maximal type. This means that $F(t)$ has to be the maximal type, which in turn implies that the corresponding premise of *f_poly_sub$_2$* will also succeed by the same rule.

- using the deterministic form of the transitivity rule, which basically proceeds by trying to prove that the bound of the variable t in the environment (*i.e.* $G(t)$) is a subtype of $F(t)$. But this is exactly the premise of $f\_poly\_sub_2$.

Hence, the only cases where $f\_poly\_sub_2$ doesn't seem sufficient is when one has types of the form $\forall t{<:}t.T$. It might even make sense to disallow such forms of bounded polymorphism.

Claim

The correct rule for subtyping F-bounded polymorphic types is

$$\frac{\Gamma \cup \{t{<:}G(t)\} \vdash G(t) <: F(t) \qquad \Gamma \cup \{t{<:}G(t)\} \vdash T_1 <: T_2}{\Gamma \vdash \forall t{<:}F(t).T_1 <: \forall t{<:}G(t).T_2}$$

Pinch of salt: Our argument was based on the simple calculus $F_{<:}$. Systems with more complicated types could involve other means of reasoning about subtyping with type variables, and our argument might fall short. Hence, we cannot rule out systems that require the more flexible subtyping rule:

$$\frac{\Gamma \cup \{t{<:}G(t)\} \vdash t <: F(t) \qquad \Gamma \cup \{t{<:}G(t)\} \vdash T_1 <: T_2}{\Gamma \vdash \forall t{<:}F(t).T_1 <: \forall t{<:}G(t).T_2}$$

References

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[CG91] P.-L. Curien and G. Ghelli. Coherence of subsumption. In *Mathematical Structures in Computer Science*. 1991.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Ghe92] Giorgio Ghelli. Divergence of $F_{\leq}$ type checking. 1992.

[KS92] D. Katiyar and S. Sankar. Completely bounded quantification is decidable. In *ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[Pie92] Benjamin Pierce. Bounded quantification is undecidable. In *Proc. 19-th ACM Symp. on Principles of Programming Languages*, pages 305–315, 1992.