Bounded Polymorphism

Weifeng Han, Fred Fu

Introduction

In the study of programming languages, bounded polymorphism is a feature created from combining subtyping with polymorphism, which are two core features of a expressive type system.

What Problem Does Bounded Polymorphism Solve?

In a type system, if subtyping and polymorphism work individually, the type system could cause the loss of some type information while typechecking certain functions. The following is an instance given in the book called *Types and Programming Languages*.[2]

```
;; Given the identity function that takes a
;; record of type {a:Nat}, it should return
;; the same type as its input, i.e., {a:Nat}
f : {a:Nat} → {a:Nat}
f = λx:{a:Nat}. x

;; Let's define a record called rab to it.
;; Since rab is a subtype of {a:Nat}, it
;; is a valid input
rab : {a:Nat, b:Nat}
rab = {a=0, b=1}

;; Now, we apply f to rab, which returns rab
;; according to the function definition, but
;; now rab is of type {a:Nat}
(f rab) = rab : {a:Nat}

;; In other words, (f rab).b is not accessible
;; because record of type {a:Nat} does not
;; guarantee to have the field called b!!
```

How Does Bounded Polymorphism solve the problem?

There are two goals that we want to achieve from the previous example.

1. Constrain the input to be a subtype of {a:Nat}.
2. Maintain the input type while type checking the function

Subtyping is meant to satisfy the first goal, and polymorphism has the power to keep the type as it is throughout the typechecking process. Thus, continuing from the previous example, we combine subtyping and polymorphism in the following way so that typing information is lost during typechecking and the constraint is still enforced.

```
;; The function definition stays the same,
;; whereas we revised the type annotation of
;; this function. It reads like the following:
;; "for any type X that is a subtype of {a:Nat},
;; function f takes a term of type X and return
;; a term of type X."
f : ∀X <: {a:Nat}. X → X
f = λx:{a:Nat}. x

;; rab stays the same
rab : {a:Nat, b:Nat}
rab = {a=0, b=1}

;; Again, we apply f to rab. According to the
;; type annotation of function f, the return
;; type is {a:Nat, b:Nat}, which means we didn't
;; lose any information unlike the previous one.
;; So, it is now valid to call (f rab).b.
(f rab) = rab : {a:Nat, b:Nat}
```

Further Discussion

Bounded polymorphisms still have limitations on recursive types such as the type of Object in object-oriented Programming. So, Canning, Cook, Hill, and Olthoff proposed the F-bounded polymorphism as the resolution to typecheck recursive types.[1] With the F-bound expression, the type system will have the power to typecheck any type that satisfies the following:

$$\forall t \subseteq F[t].\sigma$$

More details are explained in the paper [1].

Reference

[1] Canning, Peter, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. "F-bounded polymorphism for object-oriented programming." In *Proceedings of the fourth international conference on functional programming languages and computer architecture*, pp. 273-280. 1989.

[2] Pierce, Benjamin C. "Bounded Quantification." *Types and programming languages*, 389-405. MIT press, 2002.