CPSC 311: Subtyping for refs **(DRAFT)**
("lec-subtyping-ref")

Joshua Dunfield
University of British Columbia

November 15, 2016

1 Subtyping, so far

$\boxed{A <: B}$ Type A is a subtype of type B

$$\frac{}{A <: A}\text{Sub-refl} \qquad \frac{A1 <: A2 \qquad A2 <: A3}{A1 <: A3}\text{Sub-trans} \qquad \frac{}{\mathsf{pos} <: \mathsf{int}}\text{Sub-pos-int} \qquad \frac{}{\mathsf{int} <: \mathsf{rat}}\text{Sub-int-rat}$$

$$\frac{A1 <: B1 \qquad A2 <: B2}{(A1 * A2) <: (B1 * B2)}\text{Sub-product} \qquad \frac{B1 <: A1 \qquad A2 <: B2}{(A1 \to A2) <: (B1 \to B2)}\text{Sub-arr}$$

2 Typing for refs

$\boxed{\Gamma \vdash e : A}$ Under assumptions Γ, expression e has type A

$$\frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\text{Type-sub} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash (\mathsf{Id}\ x) : A}\text{Type-var}$$

$$\frac{}{\Gamma \vdash (\mathsf{Num}\ n) : \mathsf{num}}\text{Type-num} \qquad \frac{op : A1 * A2 \to B \qquad \Gamma \vdash e1 : A1 \qquad \Gamma \vdash e2 : A2}{\Gamma \vdash (\mathsf{Binop}\ op\ e1\ e2) : B}\text{Type-binop}$$

$$\frac{}{\Gamma \vdash (\mathsf{Bfalse}) : \mathsf{bool}}\text{Type-false} \qquad \frac{}{\Gamma \vdash (\mathsf{Btrue}) : \mathsf{bool}}\text{Type-true}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash eThen : A \qquad \Gamma \vdash eElse : A}{\Gamma \vdash (\mathsf{Ite}\ e\ eThen\ eElse) : A}\text{Type-ite}$$

$$\frac{x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Lam}\ x\ A\ eBody) : A \to B}\text{Type-lam} \qquad \frac{\Gamma \vdash e1 : A \to B \qquad \Gamma \vdash e2 : A}{\Gamma \vdash (\mathsf{App}\ e1\ e2) : B}\text{Type-app}$$

$$\frac{\Gamma \vdash e1 : A1 \qquad \Gamma \vdash e2 : A2}{\Gamma \vdash (\mathsf{Pair}\ e1\ e2) : A1 * A2}\text{Type-pair} \qquad \frac{\Gamma \vdash e : A1 * A2 \qquad x1 : A1, x2 : A2, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Pair\text{-}case}\ e\ x1\ x2\ eBody) : B}\text{Type-pair-case}$$

$$\frac{\Gamma \vdash e : A \qquad x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{Let}\ x\ e\ eBody) : B}\text{Type-with} \qquad \frac{u : B, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Rec}\ u\ B\ e) : B}\text{Type-rec}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash (\mathsf{Ref}\ e) : \mathsf{ref}\ A}\text{Type-ref} \qquad \frac{\Gamma \vdash e : \mathsf{ref}\ A}{\Gamma \vdash (\mathsf{Deref}\ e) : A}\text{Type-deref} \qquad \frac{\Gamma \vdash e1 : \mathsf{ref}\ A \qquad \Gamma \vdash e2 : A}{\Gamma \vdash (\mathsf{Setref}\ e1\ e2) : A}\text{Type-setref}$$

Following the pattern of product types, we might write a covariant rule for references:

$$\frac{A <: B}{(\mathsf{ref}\ A) <: (\mathsf{ref}\ B)}\ \text{??Sub-ref}$$

By this rule, $(\mathsf{ref\ int}) <: (\mathsf{ref\ rat})$. However, if you expect something of type ref rat and I give you an expression of type $(\mathsf{ref\ int})$, you can use Setref to replace the reference's contents with 3.5 (because, to you, it is a ref rat and you can assign any rat to it).

So we might try contravariance:

$$\frac{B <: A}{(\mathsf{ref}\ A) <: (\mathsf{ref}\ B)}\ \text{??Sub-ref-2}$$

Now, however, if you expect something of type $(\mathsf{ref\ int})$ and Deref it, expecting an int, you may be disappointed: By ??Sub-ref-2, $(\mathsf{ref\ rat}) <: (\mathsf{ref\ int})$. But the contents of $(\mathsf{ref\ rat})$ could be 3.5 or any rational number, not necessarily an integer.

The covariant rule ??Sub-ref works fine with Deref, but not with Setref; the contravariant rule ??Sub-ref-2 works fine with Setref, but not with Deref. So the covariant rule enforces a necessary condition for Deref, and the contravariant rule enforces a necessary condition for Setref. Therefore, a correct rule is:

$$\frac{A <: B \qquad B <: A}{(\mathsf{ref}\ A) <: (\mathsf{ref}\ B)}\ \text{Sub-ref}$$

which enforces *both* conditions.

(We might try to "optimize" this rule by replacing the premises with $A = B$. That's probably okay for this type system, but doesn't work for all type systems, so I'd rather leave it as is.)

The following may be a useful additional explanation, particularly if you understand contravariant subtyping for function types $A1 \rightarrow A2$. We can think of a reference as an object with two methods, called Deref and Setref:

- The Deref "method" has no arguments (we are thinking of this, for the moment, as a class method, so the reference to "self" or "this" is implicit), and returns (for a reference of type $(\mathsf{ref}\ A)$) a value of type A.

  So we can think of the type of Deref as $() \rightarrow A$, where $()$ represents taking zero arguments.

- The Setref "method" takes one argument, of type A (assuming the reference has type $(\mathsf{ref}\ A)$). It also returns the value of the argument. So we can think of the type of Setref as $A \rightarrow A$.

Thus, the Deref "method" has type $() \rightarrow A$ and Setref has type $A \rightarrow A$. According to the contravariant rule for functions, Sub-arr, we can compare the types of the Deref method of a reference of type $(\mathsf{ref}\ A)$ and the Deref method of a reference of type $(\mathsf{ref}\ B)$ as follows:

$$\frac{() <: () \qquad A <: B}{\big(() \rightarrow A\big) <: \big(() \rightarrow B\big)}\ \text{Sub-arr}$$

The second premise here matches the covariant premise of Sub-ref. (Regardless of whatever () is, exactly, the first premise is derivable using Sub-refl.)

For Setref, we get

$$\frac{\boxed{B <: A} \qquad A <: B}{(A \to A) <: (B \to B)}\ \text{Sub-arr}$$

The second premise here is something of an accident: we happened to decide that Setref should return the new contents just written to the reference. If we said, instead, that Setref returned "nothing", which we seem to be writing as (), then we would have

$$\frac{\boxed{B <: A} \qquad () <: ()}{(A \to ()) <: (B \to ())}\ \text{Sub-arr}$$

2.1 Upper bounds

Something I hadn't thought of by Monday's lecture: there are a few more places where we need to use Type-sub. We need to use it in Type-ite; otherwise, `typeof` will return `false` for the expression

$$(\mathsf{Ite}\ (\mathsf{Btrue})\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ -1))$$

This is because $(\mathsf{Num}\ 1)$ has type pos, and $(\mathsf{Num}\ -1)$ has type int, but pos $\neq$ int. So when we implement Type-ite, we need to find the *upper bound* of the types of the $eThen$ and $eElse$ branches:

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash eThen : A \qquad \Gamma \vdash eElse : A}{\Gamma \vdash (\mathsf{Ite}\ e\ eThen\ eElse) : A}\ \text{Type-ite}$$

$$\frac{\Gamma \vdash e : B \qquad B = \mathsf{bool} \qquad \Gamma \vdash eThen : A1 \qquad \Gamma \vdash eElse : A2 \qquad A1 = A2}{\Gamma \vdash (\mathsf{Ite}\ e\ eThen\ eElse) : A1}\ \text{Type-ite}$$

$$\frac{\Gamma \vdash e : B \qquad B <: \mathsf{bool} \qquad \Gamma \vdash eThen : A1 \qquad \boxed{A1 <: A} \qquad \Gamma \vdash eElse : A2 \qquad \boxed{A2 <: A}}{\Gamma \vdash (\mathsf{Ite}\ e\ eThen\ eElse) : \boxed{A}}\ \text{Type-ite*}$$

This last version of Type-ite, marked *, is really just the original Type-ite with three uses of Type-sub:

$$\frac{\dfrac{\Gamma \vdash e : B \qquad B <: \mathsf{bool}}{\Gamma \vdash e : \mathsf{bool}}\ \text{Type-sub} \qquad \dfrac{\Gamma \vdash eThen : A1 \qquad A1 <: A}{\Gamma \vdash eThen : A}\ \text{Type-sub} \qquad \dfrac{\Gamma \vdash eElse : A2 \qquad A2 <: A}{\Gamma \vdash eElse : A}\ \text{Type-sub}}{\Gamma \vdash (\mathsf{Ite}\ e\ eThen\ eElse) : A}\ \text{Type-ite}$$

That is, Type-ite* is an easier rule to implement, but Type-ite* isn't adding any power to the type system. (It's harder, actually, to prove that Type-ite* isn't *taking anything away* from the type system. But I'm pretty sure it isn't.)

I wrote a function `upper-bound` that takes two types A and B, and returns A if $B <: A$, and B if $A <: B$. See `a5.rkt`.