

Type Theory (15-814) Fall 2006

Homework 3: Subtyping

William Lovas (wlovas@cs)

Out: Saturday, October 28, 2006

Due: Thursday, November 9, 2006 (before 1:30 pm)

In this homework, you'll explore several aspects of subtyping, including:

- its expressive power,
- its effect on our theorems, and
- a coercion interpretation that eliminates subtyping.

You should submit two files, `solution.pdf` and `typecheck.sml`. As usual, you can do so by copying them to the directory

`/afs/cs.cmu.edu/academic/class/15814/handin/<userid>/hw3`

This assignment is a bit heavier on programming than the previous two, so be careful not to get caught off guard.

Good luck!

1 Theory: Subtyping

For this homework, we'll be working with a variation of PCF with records and variants (n -ary labeled products and sums, respectively), subtyping, and recursive types. The syntax of this language, which we'll refer to as $\text{PCF}_{<}^{\mu}$, is shown below.

$$\begin{aligned} \tau &::= X \mid \tau_1 \rightarrow \tau_2 \mid \{ \ell_i : \tau_i \}_{i=1}^n \mid [\ell_i : \tau_i]_{i=1}^n \mid \mu X. \tau \\ e &::= x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \{ \ell_i = e_i \}_{i=1}^n \mid e. \ell \mid [\ell = e]_{\tau} \mid \mathbf{case}_{\tau} e \{ [\ell_i = x_i] \Rightarrow e_i \}_{i=1}^n \mid \mathbf{roll}_{\tau} e \mid \mathbf{unroll} e \end{aligned}$$

The declarative typing rules for this language appear in Figures 1 and 2. In Figure 2, the rule $S\text{-}\{\}\text{-WIDTH}$ is the record subtyping rule given in lecture, and the rule $S\text{-}\{\}\text{-DEPTH}$ states that records fields are subtyped covariantly, by analogy with products. The rule $S\text{-VAR}$ uses a subtyping assumption from the subtyping context Σ ; the rule $S\text{-}\mu$ adds such an assumption in its premise.

1.1 Warmup: Hacking in $\text{PCF}_{<}^{\mu}$

As with many languages we study in this course, $\text{PCF}_{<}^{\mu}$'s austerity is deceiving—these few constructs give rise to a rich theory of programming similar in style to ML. For example, the type `nat` included as a primitive in previous iterations of PCF is definable in $\text{PCF}_{<}^{\mu}$ as follows:

$$\begin{aligned} \mathbf{nat} &::= \mu N. [\mathbf{zero} : \mathbf{unit}, \mathbf{succ} : N] \\ \mathbf{zero} &::= \mathbf{roll}_{\mathbf{nat}} [\mathbf{zero} = \langle \rangle] \\ \mathbf{succ} e &::= \mathbf{roll}_{\mathbf{nat}} [\mathbf{succ} = e], \\ \mathbf{natcase}_{\tau} e \{ \mathbf{zero} \Rightarrow e_0, \mathbf{succ} x \Rightarrow e_1 \} &::= \mathbf{case}_{\tau} \mathbf{unroll} e \{ [\mathbf{zero} = _] \Rightarrow e_0, [\mathbf{succ} = x] \Rightarrow e_1 \} \end{aligned}$$

$\Gamma \vdash e : \tau$

$\frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x:\tau_1. e : \tau_1 \rightarrow \tau_2} (\rightarrow\text{-I})$	$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} (\rightarrow\text{-E})$	
$\frac{(\Gamma \vdash e_i : \tau_i)_{i=1}^n}{\Gamma \vdash \{\ell_i = e_i\}_{i=1}^n : \{\ell_i : \tau_i\}_{i=1}^n} (\{\}\text{-I})$	$\frac{\Gamma \vdash e : \{\ell_i : \tau_i\}_{i=1}^n}{\Gamma \vdash e. \ell_j : \tau_j} (\{\}\text{-E})$	
$\frac{\Gamma \vdash e : \tau_j}{\Gamma \vdash [\ell_j = e]_{[\ell_i : \tau_i]_{i=1}^n} : [\ell_i : \tau_i]_{i=1}^n} ([\]\text{-I})$	$\frac{\Gamma \vdash e : [\ell_i : \tau_i]_{i=1}^n \quad (\Gamma, x_i:\tau_i \vdash e_i : \tau)_{i=1}^n}{\Gamma \vdash \mathbf{case}_\tau e \{[\ell_i = x_i] \Rightarrow e_i\}_{i=1}^n : \tau} ([\]\text{-E})$	
$\frac{\Gamma \vdash e : [\mu X. \tau/X] \tau}{\Gamma \vdash \mathbf{roll}_{\mu X. \tau} e : \mu X. \tau} (\mu\text{-I})$	$\frac{\Gamma \vdash e : \mu X. \tau}{\Gamma \vdash \mathbf{unroll} e : [\mu X. \tau/X] \tau} (\mu\text{-E})$	
$\frac{\Gamma, x:\tau \vdash c : \tau}{\Gamma \vdash \mathbf{fix} x:\tau. c : \tau} (\text{FIX})$	$\frac{}{\Gamma, x:\tau \vdash x : \tau} (\text{VAR})$	$\frac{\Gamma \vdash e : \tau' \quad \cdot \vdash \tau' <: \tau}{\Gamma \vdash e : \tau} (\text{SUB})$

Figure 1: Declarative typing for $\text{PCF}_{<}^\mu$.

$\Sigma \vdash \sigma <: \tau$

$\frac{}{\Sigma \vdash \{\ell_i : \tau_i\}_{i=1}^{n+k} <: \{\ell_i : \tau_i\}_{i=1}^n} (\text{S-}\{\}\text{-WIDTH})$	$\frac{(\Sigma \vdash \sigma_i <: \tau_i)_{i=1}^n}{\Sigma \vdash \{\ell_i : \sigma_i\}_{i=1}^n <: \{\ell_i : \tau_i\}_{i=1}^n} (\text{S-}\{\}\text{-DEPTH})$	
$\frac{}{\Sigma \vdash [\ell_i : \tau_i]_{i=1}^n <: [\ell_i : \tau_i]_{i=1}^{n+k}} (\text{S-}[\]\text{-WIDTH})$	$\frac{(\Sigma \vdash \sigma_i <: \tau_i)_{i=1}^n}{\Sigma \vdash [\ell_i : \sigma_i]_{i=1}^n <: [\ell_i : \tau_i]_{i=1}^n} (\text{S-}[\]\text{-DEPTH})$	
$\frac{\Sigma \vdash \tau <: \sigma \quad \Sigma \vdash \sigma' <: \tau'}{\Sigma \vdash \sigma \rightarrow \sigma' <: \tau \rightarrow \tau'} (\text{S-}\rightarrow)$	$\frac{\Sigma, X <: Y \vdash \sigma <: \tau \quad X \# Y \quad X \# \Sigma \quad Y \# \Sigma}{\Sigma \vdash \mu X. \sigma <: \mu Y. \tau} (\text{S-}\mu)$	
$\frac{}{\Sigma, X <: Y \vdash X <: Y} (\text{S-VAR})$	$\frac{}{\Sigma \vdash \tau <: \tau} (\text{S-REFL})$	$\frac{\Sigma \vdash \sigma <: \rho \quad \Sigma \vdash \rho <: \tau}{\Sigma \vdash \sigma <: \tau} (\text{S-TRANS})$

Figure 2: Declarative subtyping for $\text{PCF}_{<}^\mu$.

where the type `unit` is just the empty record type `{ }` and the null tuple `()` is just the empty record `{ }`. **Note:** we've elided the type annotations on the variant injections for brevity.

Following this pattern, one can define ML-like datatypes such as lists and trees.

Exercise 1. Give a definition of the `natlist` datatype by completing the following template:

$$\begin{aligned} \text{natlist} &:= \mu L. [\text{nil} : \text{unit}, \text{cons} : \text{nat} \times L] \\ \text{nil} &:= \dots \\ \text{cons}(e_1, e_2) &:= \dots \\ \mathbf{listcase}_\tau e \{ \text{nil} \Rightarrow e_{\text{nil}}, \text{cons}(x, xs) \Rightarrow e_{\text{cons}} \} &:= \dots \end{aligned}$$

You may assume an encoding of natural numbers `nat`, products $\tau_1 \times \tau_2$, and the unit type. Your definition should satisfy the following specification:

$$\begin{aligned} \mathbf{listcase}_\tau \text{nil} \{ \text{nil} \Rightarrow e_{\text{nil}}, \text{cons}(x, xs) \Rightarrow e_{\text{cons}} \} &\mapsto^* e_{\text{nil}} \\ \mathbf{listcase}_\tau \text{cons}(e_1, e_2) \{ \text{nil} \Rightarrow e_{\text{nil}}, \text{cons}(x, xs) \Rightarrow e_{\text{cons}} \} &\mapsto^* [e_1/x] [e_2/xs] e_{\text{cons}} \end{aligned}$$

A common optimization for the representation of lists is to allow the `cons`'ing of multiple elements onto the front at once; this technique is sometimes called "CDR-coding", and it can reduce memory requirements by using roughly half as many pointers.

Exercise 2. Consider a $\text{PCF}_{<}^\mu$ type for CDR-coded lists,

$$\text{natlist2} := \mu L. [\text{nil} : \text{unit}, \text{cons} : \text{nat} \times L, \text{cons2} : \text{nat} \times \text{nat} \times L]$$

Show by means of a subtyping derivation that $\cdot \vdash \text{natlist} <: \text{natlist2}$. (You may make use of the covariance rule for products,

$$\frac{\Sigma \vdash \sigma_1 <: \tau_1 \quad \Sigma \vdash \sigma_2 <: \tau_2}{\Sigma \vdash \sigma_1 \times \sigma_2 <: \tau_1 \times \tau_2} \text{(S-X-DEPTH)}$$

Given the appropriate encoding of $\tau_1 \times \tau_2$, it is derivable.)

1.2 Type safety and subtyping

Suppose we're trying to prove type safety, i.e. Progress and Preservation, for $\text{PCF}_{<}^\mu$. Recall that we prove Preservation by induction on evaluation $e \mapsto e'$ (not shown in this handout). We might proceed as follows:

Case: $\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2}$

$\Gamma \vdash e_1 e_2 : \tau$
 $\Gamma \vdash e_1 : \tau_2 \rightarrow \tau$ and $\Gamma \vdash e_2 : \tau_2$?

By assumption
 By Inversion?

The proof fails at this step because our old Inversion theorem no longer holds: it was proven by observing that the only rule that could type an expression of the form $e_1 e_2$ was \rightarrow -E, so the conclusion implied the premises. But now, in the presence of subtyping, there's another rule that could have concluded $\Gamma \vdash e_1 e_2 : \tau$, namely the subsumption rule `SUB`.

Exercise 3. State and prove an appropriate lemma that allows the proof of Preservation to continue. Then use your lemma to finish this case of the Preservation theorem.

$\Sigma \vdash \sigma \triangleleft: \tau$

$$\begin{array}{c}
\frac{}{\Sigma, X \triangleleft: Y \vdash X \triangleleft: Y} \text{(AS-VAR)} \quad \frac{(\Sigma \vdash \sigma_i \triangleleft: \tau_i)_{i=1}^n}{\Sigma \vdash \{\ell_i : \sigma_i\}_{i=1}^{n+k} \triangleleft: \{\ell_i : \tau_i\}_{i=1}^n} \text{(AS-{})} \quad \frac{(\Sigma \vdash \sigma_i \triangleleft: \tau_i)_{i=1}^n}{\Sigma \vdash [\ell_i : \sigma_i]_{i=1}^n \triangleleft: [\ell_i : \tau_i]_{i=1}^{n+k}} \text{(AS-[])} \\
\\
\frac{\Sigma \vdash \tau \triangleleft: \sigma \quad \Sigma \vdash \sigma' \triangleleft: \tau'}{\Sigma \vdash \sigma \rightarrow \sigma' \triangleleft: \tau \rightarrow \tau'} \text{(AS-}\rightarrow\text{)} \quad \frac{\Sigma, X \triangleleft: Y \vdash \sigma \triangleleft: \tau \quad X \# Y \quad X \# \Sigma \quad Y \# \Sigma}{\Sigma \vdash \mu X. \sigma \triangleleft: \mu Y. \tau} \text{(AS-}\mu\text{)}
\end{array}$$

Figure 3: Algorithmic subtyping for $\text{PCF}_{<}^{\mu}$.

1.3 Coercion semantics preliminaries

One way to interpret a subtyping judgement $\Sigma \vdash \sigma <: \tau$ is that there exists a coercion from σ to τ , i.e. a function of type $\sigma \rightarrow \tau$. By interpreting the subtyping judgement this way, one can transform a term from a language with subtyping into a term from one without subtyping by inserting coercions into the term everywhere subtyping was required.

For example, in $\text{PCF}_{<}^{\mu}$ we might have the following term:

$$(\lambda x : \{a : \text{nat}\}. x.a) \{a = 5, b = 7\}$$

In order to type this term, we need subsumption to tell us that the record with an “a” and a “b” field can be thought of as a record with just an “a” field. Under the coercion interpretation, we would insert a function to realize the subsumption by dropping the “b” field:

$$(\lambda x : \{a : \text{nat}\}. x.a) \left((\lambda r : \{a : \text{nat}, b : \text{nat}\}. \{a = r.a\}) \{a = 5, b = 7\} \right)$$

This new term can be typed without using the `SUB` rule.

As discussed in class, one problem with the coercion interpretation is that there may in general be many ways of deriving that $\cdot \vdash \sigma <: \tau$ for two given types σ and τ , which means that there may be many coercion functions. We’d like to know that all of those coercions are equivalent. One easy way to do this is to force there to be only one subtyping derivation for each judgement $\cdot \vdash \sigma <: \tau$, and a key step in this endeavor is eliminating the transitivity rule.

Exercise 4. Is the transitivity rule `S-TRANS` in Figure 2 admissible? Can you give a counterexample, i.e. a subtyping judgement which is not derivable without using `S-TRANS`? (You need not prove that your counterexample actually requires `S-TRANS`, but you should make a convincing argument.)

Rules defining an algorithmic subtyping judgement for $\text{PCF}_{<}^{\mu}$ is shown in Figure 3. We write this judgement $\Sigma \vdash \sigma \triangleleft: \tau$ to distinguish it from the declarative version. These rules are “algorithmic” in the sense that for any pair of types in the subtype relation, there’s only one derivation concluding so. But how can we be sure these new rules really define the same relation? We’d like to show that our algorithm is *sound* and *complete* with respect to the declarative semantics: in other words, $\Sigma \vdash \sigma <: \tau$ if and only if $\Sigma \vdash \sigma \triangleleft: \tau$. Along the way, we’ll have to show that a rule similar to `S-TRANS` is admissible.

Exercise 5. Prove that transitivity is admissible under the algorithmic subtyping rules in Figure 3. That is, prove that if $\Sigma \vdash \sigma \triangleleft: \rho$ and $\Sigma \vdash \rho \triangleleft: \tau$, then $\Sigma \vdash \sigma \triangleleft: \tau$.

Hint: Try generalizing the theorem and proving a stronger result. (This may seem counterintuitive, but remember, by proving a stronger theorem, we get a stronger inductive hypothesis!)

2 Programming: Bidirectional typechecking and coercion semantics

2.1 Bidirectional typechecking

For the programming part of this assignment, we'll be working with a refinement of $\text{PCF}_{<}^{\mu}$ that supports a form of type inference called bidirectional typing. In a bidirectionally-typed system, terms are split into two syntactic categories: *synthesis* terms, s , and *checking* terms, c . The intuition is that a synthesis term s knows its type, whereas a checking term c doesn't, so it must occur in a context where its type is determined externally. To this end, we define two judgements:

- $\Gamma \vdash s \vec{\vdash} \tau$. The context Γ and the term s are inputs, and the type τ is *synthesized* as an output.
- $\Gamma \vdash c \overleftarrow{\vdash} \tau$. The context Γ , the term c , and the type τ are all inputs; if a derivation exists, then c *checks* against τ .

The direction of the arrow is intended to indicate the direction of information flow: in synthesis mode, a term produces its type; in checking mode, a type is forced onto a term.

Having stratified the system in this way, we can remove type decorations from things like λ -abstraction and variant injection, making such terms checked. We then replace the type decorations with a general type annotation construct, $(c : \tau)$, which turns a checking term into a synthesis one.

$$\begin{aligned} \tau &::= X \mid \tau_1 \rightarrow \tau_2 \mid \{ \ell_i : \tau_i \}_{i=1}^n \mid [\ell_i : \tau_i]_{i=1}^n \mid \mu X. \tau \\ s &::= x \mid s \ c \mid s. \ell \mid \mathbf{unroll} \ s \mid (c : \tau) \\ c &::= s \mid \lambda x. c \mid \{ \ell_i = c_i \}_{i=1}^n \mid [\ell = c] \mid \mathbf{case} \ s \{ [\ell_i = x_i] \Rightarrow c_i \}_{i=1}^n \mid \mathbf{roll} \ c \mid \mathbf{fix} \ x. c \end{aligned}$$

Variables are synthesizing terms, since their type is given in the context. λ -abstraction is a checked term, since we can't know the type of its argument without a type decoration. All synthesizing terms can be thought of as checked terms, since we can synthesize their type and compare with the type we're checking it at.

Note that there is some freedom in deciding where to place certain terms. For example, records could easily be synthesis terms, but we've chosen to make them checked. The general idea behind the stratification is that type annotations should appear only where introduction forms meet elimination forms.

The rules defining synthesis and checking appear in Figure 4.

Bidirectional typing is a form of type inference, in that it allows us to omit many needless type annotations, but it is *local* in the sense that it only propagates type information between adjacent nodes in the abstract syntax tree. This is in direct contrast to unification-based inference methods as seen in languages like ML and Haskell; unification propagates type constraints in a global fashion, making it very hard to produce a typechecker with good error messages.

Program 1. Write a bidirectional typechecker for $\text{PCF}_{<}^{\mu}$, using the rules in Figure 4 as a guide.

2.2 Coercion semantics

As it stands, the bidirectional language we've defined admits no subtyping—really, it's more like PCF^{μ} than $\text{PCF}_{<}^{\mu}$. Interestingly, to add support for subtyping in a completely syntax-directed way, we only need to change one rule: the CHK-CHECKSYNTH rule, where we switch modes from checking a type to synthesizing a type. Instead of demanding that the type we synthesize be *equal* to the type we're checking at, we permit it to be more precise, that is, a subtype of the checked type. The relevant rule calls out to algorithmic subtyping to achieve this end, as shown in Figure 5.

Using the algorithmic rules given above, we can code up a translation from PCF^{μ} terms to $\text{PCF}_{<}^{\mu}$ terms by inserting appropriate coercions, as discussed earlier. To do so, we employ a judgement based on algorithmic subtyping, written $\Sigma \vdash \sigma \triangleleft : \tau \rightsquigarrow s$. This judgement is just like algorithmic subtyping, but it produces a coercion term s that witnesses the subtyping from σ to τ . (Note: the judgement outputs a synthesizing term

$\Gamma \vdash s \vec{\cdot} \tau$

$$\frac{\Gamma \vdash s \vec{\cdot} \tau_2 \rightarrow \tau \quad \Gamma \vdash c \overleftarrow{\cdot} \tau_2}{\Gamma \vdash s c \vec{\cdot} \tau} \text{ (SYN-}\rightarrow\text{-E)}$$

$$\frac{\Gamma \vdash s \vec{\cdot} \{\ell_i : \tau_i\}_{i=1}^n}{\Gamma \vdash s. \ell_j \vec{\cdot} \tau_j} \text{ (SYN-}\{\cdot\}\text{-E)}$$

$$\frac{\Gamma \vdash s \vec{\cdot} \mu X. \tau}{\Gamma \vdash \mathbf{unroll} s \vec{\cdot} [\mu X. \tau / X] \tau} \text{ (SYN-}\mu\text{-E)}$$

$$\frac{\Gamma \vdash c \overleftarrow{\cdot} \tau}{\Gamma \vdash (c : \tau) \vec{\cdot} \tau} \text{ (SYN-SYNTHCHECK)}$$

$\Gamma \vdash c \overleftarrow{\cdot} \tau$

$$\frac{\Gamma \vdash s \vec{\cdot} \tau}{\Gamma \vdash s \overleftarrow{\cdot} \tau} \text{ (CHK-CHECKSYNTH)}$$

$$\frac{\Gamma, x \vec{\cdot} \tau_1 \vdash c \overleftarrow{\cdot} \tau_2}{\Gamma \vdash \lambda x. c \overleftarrow{\cdot} \tau_1 \rightarrow \tau_2} \text{ (CHK-}\rightarrow\text{-I)}$$

$$\frac{(\Gamma \vdash c_i \overleftarrow{\cdot} \tau_i)_{i=1}^n}{\Gamma \vdash \{\ell_i = c_i\}_{i=1}^n \overleftarrow{\cdot} \{\ell_i : \tau_i\}_{i=1}^n} \text{ (CHK-}\{\cdot\}\text{-I)}$$

$$\frac{\Gamma \vdash c \overleftarrow{\cdot} \tau_j}{\Gamma \vdash [\ell_j = c] \overleftarrow{\cdot} [\ell_i : \tau_i]_{i=1}^n} \text{ (CHK-}\{\cdot\}\text{-I)}$$

$$\frac{\Gamma \vdash s \vec{\cdot} [\ell_i : \tau_i]_{i=1}^n \quad (\Gamma, x_i \vec{\cdot} \tau_i \vdash c_i \overleftarrow{\cdot} \tau)_{i=1}^n}{\Gamma \vdash \mathbf{case} s \{ [\ell_i = x_i] \Rightarrow c_i \}_{i=1}^n \overleftarrow{\cdot} \tau} \text{ (CHK-}\{\cdot\}\text{-E)}$$

$$\frac{\Gamma \vdash c \overleftarrow{\cdot} [\mu X. \tau / X] \tau}{\Gamma \vdash \mathbf{roll} c \overleftarrow{\cdot} \mu X. \tau} \text{ (CHK-}\mu\text{-I)}$$

$$\frac{\Gamma, x \vec{\cdot} \tau \vdash c \overleftarrow{\cdot} \tau}{\Gamma \vdash \mathbf{fix} x. c \overleftarrow{\cdot} \tau} \text{ (CHK-FIX)}$$

Figure 4: Bidirectional typing for $\text{PCF}_{<}^{\mu}$

$\Gamma \vdash c \overleftarrow{\cdot} \tau$

$$\frac{\Gamma \vdash s \vec{\cdot} \tau' \quad \cdot \vdash \tau' \triangleleft \tau}{\Gamma \vdash s \overleftarrow{\cdot} \tau} \text{ (CHK-CHECKSYNTH')} \quad \dots$$

Figure 5: Bidirectional $\text{PCF}_{<}^{\mu}$ with algorithmic subtyping

rather than a checking term since we always want to apply the coercion, and only synthesizing terms can appear in the function position of an application.) The relevant theorem would be something like this:¹

Theorem. If $\Sigma \vdash \sigma \triangleleft: \tau \rightsquigarrow s$, then $\Gamma(\Sigma) \vdash s \overset{\rightarrow}{:} \sigma \rightarrow \tau$.

$\Gamma(\Sigma)$ represents a typing context appropriate to the subtyping context Σ . Let us defer the definition of $\Gamma(\Sigma)$ for the moment to see an example.

Above, we showed how a coercion for records could drop fields in order to reflect width subtyping. Our algorithmic rule includes both depth subtyping too,, so we have to do a little more than just drop fields. But the premise of AS-{} will give us the appropriate depth subtyping coercions, so we just need to apply them. The rule looks like this:

$$\frac{\left(\Sigma \vdash \sigma_i \triangleleft: \tau_i \rightsquigarrow s_i\right)_{i=1}^n}{\Sigma \vdash \{\ell_i : \sigma_i\}_{i=1}^{n+k} \triangleleft: \{\ell_i : \tau_i\}_{i=1}^n \rightsquigarrow (\lambda r. \{\ell_i = s_i (r.\ell_i)\}_{i=1}^n : \{\ell_i : \sigma_i\}_{i=1}^{n+k} \rightarrow \{\ell_i : \tau_i\}_{i=1}^n)} \text{(COERCE-}\{\}\text{)}$$

In words, given a record with $n + k$ fields of types σ_i , we create a new record with n fields of types τ_i by projecting the relevant σ_i 's out of the original record and applying coercion functions s_i to turn them into τ_i 's. Finally, we have to add an annotation to turn the checking λ -term into a synthesis term. You can easily verify that the above theorem holds for this rule.

What about variable types? When we add a subtyping assumption $X \triangleleft: Y$ to Σ , what can its witness possibly be? Since we're *assuming* that X is a subtype of Y , we must also assume the existence of a coercion function from X to Y . In other words, we bind variable names to our subtyping assumptions in Σ , just like we would in Γ . Then the variable rule is trivial:

$$\frac{}{\Sigma, z: X \triangleleft: Y \vdash X \triangleleft: Y \rightsquigarrow z} \text{(COERCE-VAR)}$$

Now we're prepared to define $\Gamma(\Sigma)$, the typing context resulting from a (variable-annotated) subtyping context Σ :

$$\begin{aligned} \Gamma(\cdot) &:= \cdot \\ \Gamma(\Sigma, z: X \triangleleft: Y) &:= \Gamma(\Sigma), z: X \rightarrow Y \end{aligned}$$

Exercise 6. Complete the definition of $\Sigma \vdash \sigma \triangleleft: \tau \rightsquigarrow s$ by writing rules COERCE-[], COERCE- \rightarrow , and COERCE- μ . Let the above theorem guide you as you design your rules. Include them in your solution.pdf.

Program 2. Implement your coercion semantics as a function:

```
subtype : Syntax.ty * Syntax.ty -> Syntax.synth_exp option
```

Then, based on your typechecker write a function:

```
elabSynthExp : Syntax.synth_exp -> Syntax.synth_exp * Syntax.ty
```

The case for ChkSynth expressions should call `subtype` to produce a coercion and then output a `SynApp` term that applies the coercion to the original term. Any term output by `elabSynthExp` should synthesize a type in your original bidirectional typechecker without subtyping.

See the file `typecheck-sig.sml` for more info. You'll find some useful utility functions in the `Util` module, whose signature is defined in `util-sig.sml`.

¹"Something like" rather than "precisely" because to be precise, we'd have to take into account the type variables in play via a context Δ , similarly to how we do with System F.