Assignment 6: Coercion Elaboration

15-312: Foundations of Programming Languages
Kevin Watkins ⟨kw@cmu.edu⟩

Out: Tuesday, March 29, 2005
Due: Thursday, April 7, 2005 (11:59 p.m.)

100 points total

§ 1. Introduction In this assignment you will modify an implementation of PolyMinML by adding a coercion-based notion of subtyping. The language includes most of the constructs we've seen in the course, including universal, existential, and recursive types and references for mutable state.

The idea of *coercion elaboration* is to take a program that typechecks with applications of the subsumption rule, and *elaborate* it by inserting code at each point where subsumption is used, to actually perform the coercions. For example, the subsumption rule might allow an **int** expression to be used where a **float** was expected. In that case the coercion elaboration would insert the primitive operation **itof** to explicitly convert the **int** to a **float**.

Once elaboration has been done to get rid of instances of the subsumption rule, the program can be fed to the operational semantics and evaluated. If we tried to run a program without elaborating it first, it could get stuck. For example, we might imagine that integers and floating-point numbers are stored in separate registers in the machine, according to its calling convention. In that case the subsumption rule (which just says "an **int** is a **float**") needs to be replaced with actual work to move the value from one kind of register to the other.

The static semantics of our language has also been refined to get rid of the typing annotations that were formerly needed on many of the expression constructs to ensure unique typing. Instead, we use the *bidirectional* system discussed in recitation to replace all these different kinds of typing annotations with a single construct.

The text of the assignment is rather long; this is mostly because the topics here were only treated in recitation, and I also want to give you a "reference card" for almost all of the features we looked at in the static semantics of PolyMinML.

§ 2. Syntax The syntax of the variation of PolyMinML we'll be using is as follows:

$$\begin{array}{rcl}
\tau & ::= & \alpha \mid \textbf{int} \mid \textbf{bool} \mid \textbf{float} \mid \tau_1 \to \tau_2 \mid 1 \mid \tau_1 \times \cdots \times \tau_n \mid 0 \mid \tau_1 + \cdots + \tau_n \\
& & \quad \mid \tau\,\textbf{ref} \mid \tau\,\textbf{cont} \mid \forall\alpha.\,\tau \mid \exists\alpha.\,\tau \mid \mu\alpha.\,\tau \\
e & ::= & x \mid \textbf{fun}\,f(x).\,e \mid e_1\,e_2 \\
& & \quad \mid () \mid (e_1,\ldots,e_n) \mid e.\,i \\
& & \quad \mid \textbf{in}_i\,e \mid (\textbf{case}\,e\,\textbf{of}\,\textbf{in}_1\,x_1 \Rightarrow e_1 \mid \ldots \mid \textbf{in}_n\,x_n \Rightarrow e_n) \mid \textbf{abort}\,e \\
& & \quad \mid \textbf{true} \mid \textbf{false} \mid \textbf{if}\,e\,\textbf{then}\,e_1\,\textbf{else}\,e_2 \mid e_1 = e_2 \mid e_1 < e_2 \\
& & \quad \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid \tilde{}e \mid e_1\,\textbf{mod}\,e_2 \\
& & \quad \mid f \mid e_1 +.\, e_2 \mid e_1 \times.\, e_2 \mid \textbf{sqrt}\,e \mid \textbf{itof}\,e \\
& & \quad \mid \textbf{ref}\,e \mid !e \mid e_1 := e_2 \mid (e_1; e_2) \\
& & \quad \mid \textbf{letcc}\,x\,\textbf{in}\,e \mid \textbf{throw}\,e_1\,\textbf{to}\,e_2 \\
& & \quad \mid \textbf{try}\,e_1\,\textbf{ow}\,e_2 \mid \textbf{fail} \\
& & \quad \mid \Lambda\alpha.\,e \mid e[\tau] \\
& & \quad \mid \textbf{pack}(\tau, e) \mid \textbf{unpack}(\alpha, x) = e_1\,\textbf{in}\,e_2 \\
& & \quad \mid \textbf{roll}\,e \mid \textbf{unroll}\,e \\
& & \quad \mid \textbf{let}\,x = e_1\,\textbf{in}\,e_2 \mid \textbf{let type}\,\alpha = \tau\,\textbf{in}\,e \mid (e : \tau) \\
\Gamma & ::= & \cdot \mid \Gamma, (\alpha\,\textbf{type}) \mid \Gamma, (x : \tau)
\end{array}$$

(As you can see, we've put together pretty much all the concepts we've studied so far. Once you've thrown coercion-based subtyping into the mix as part of doing this assignment, you will have implemented a language that's syntactically simple, but with more subtle features than any currently marketed language, and nearly every experimental language.)

There is new syntax here, modeled on SML's, concerning references and imperative programming:

ref e	Make a new reference
$!e$	Get the value stored in e
$e_1 := e_2$	Store the value of e_2 in e_1
$(e_1; e_2)$	Evaluate e_1, then e_2

We've also modified the syntax for product types and sum types to take any number $n \geq 2$ of arguments. Although syntax for unit types (1) and void types (0) is given above, in the implementation these are reduced by the parser to special cases (with $n = 0$) of the general product and sum types. Similarly the unit expression () is turned by the parser into just a 0-ary tuple, and the abort expression **abort** e is turned by the parser into a **case** with no arms. (You should think about why an **abort** and a **case** with no arms are the same thing.) Thus, in what follows, I won't mention the unit and void types again, since the SML typechecker never has to deal with them explicitly.

The corresponding SML representations of these things now take lists:

```
datatype exp =
    ...
  | Tuple of exp list
  | Proj of exp * int
  | Inj of int * exp
  | Case of exp * (bind * exp) list
```

So for instance

$$\begin{array}{rll}
(e_1, e_2, e_3) & \text{becomes} & \texttt{Tuple[}\ulcorner e_1\urcorner\texttt{,}\ulcorner e_2\urcorner\texttt{,}\ulcorner e_3\urcorner\texttt{]} \\
e.\,5 & \text{becomes} & \texttt{Proj(}\ulcorner e\urcorner\texttt{, 5)} \\
\mathbf{in}_3\, e & \text{becomes} & \texttt{Inj(3, }\ulcorner e\urcorner\texttt{)} \\
\mathbf{case}\ e\ \mathbf{of}\ \mathbf{in}_1\, x_1 \Rightarrow e_1 & \text{becomes} & \texttt{Case(}\ulcorner e\urcorner\texttt{, [(}\ulcorner x_1\urcorner\texttt{,}\ulcorner e_1\urcorner\texttt{),} \\
\mid \mathbf{in}_2\, x_2 \Rightarrow e_2 & & \qquad\qquad\texttt{(}\ulcorner x_2\urcorner\texttt{,}\ulcorner e_2\urcorner\texttt{)])}
\end{array}$$

There is now a type **float** and operations on it:

f	A floating-point constant
$e_1 +. \, e_2$	Floating-point addition
$e_1 \times. \, e_2$	Floating-point multiplication
sqrt e	Square root

An additional operation **itof**e is used internally to do the work of coercing **int**s to **float**s. There's generally no reason to use it in the external language, because in the external language, an **int** can be used whenever a **float** is required.

A new construct **let type** $\alpha = \tau$ **in** e has been added, which lets you give a type a shorter name. This should help you if you choose to do the following extra credit problem. The operational semantics is just

$$(\mathbf{let\ type}\ \alpha = \tau\ \mathbf{in}\ e) \mapsto \{\tau/\alpha\}e$$

and the static semantics is given below.

Q 2.1. [extra credit] Code up a few of your System F examples, or an abstract type (like queues) in PolyMinML. Put them in a file `polytest.mml` and hand it in.

Finally, note that all of the type subscripts have been removed from the expression syntax. The only remaining places types can occur within an expression are when applying a type function, **pack**ing an existential (abstract) type, in the **let type** construct, and in a new expression form $(e : \tau)$ that explicitly *ascribes* the type τ to the expression e.

The ASCII syntax for all these things should be pretty clear; consult the comments at the beginning of the parser in `parse.sml` for details. As usual, a construct like **fun**, **case**, **if**, **let**, **letcc**, **throw**, **try**, **pack**, or **unpack** extends as far to the right as possible *except* that they all end at the colon in $e : \tau$ (which helps keep the parentheses for type ascriptions to a minimum). So for example we can say

```
fun f(x) is x+1 : int -> int
```

You can look at the example PolyMinML files to see how this works.

The universal, existential, and recursive types get ASCII syntax as follows:

`!a a -> a`	Read as $\forall\alpha.\,\alpha \to \alpha$
`?a a -> a`	Read as $\exists\alpha.\,\alpha \to \alpha$
`#a a -> a`	Read as $\mu\alpha.\,\alpha \to \alpha$

One other minor change in the ASCII syntax is that the expressions you type into the system (or put in a `.mml` file) have to be terminated with `;;` because a single `;` is now for sequencing imperative computations.

§ 3. Bidirectional Checking Now we'll consider how to typecheck in bidirectional style. The basic idea is to separate the single judgment $\Gamma \vdash e : \tau$ we had before into two judgments, $\Gamma \vdash e \Leftarrow \tau$ for *checking* an expression e against a type τ, and $\Gamma \vdash e \Rightarrow \tau$ for *synthesizing* (computing) a type τ by looking at the expression e. Thus we take into account that type information might flow *downward* from an expression into its subexpressions, or *upward* from the subexpressions to the whole expression. There is also, as usual, a judgment $\Gamma \vdash \tau : \textbf{type}$ declaring that the type τ is well-formed in context Γ. This doesn't need to be separated into two judgments, because we always know that we're checking τ as a *type*. There is also a judgment $\Gamma \vdash \tau_1 \sqsubseteq \tau_2$ for checking that one type is a subtype of another; here both τ_1 and τ_2 are inputs.

In the notation of *modes* we saw in recitation, we then have the following modes for the four basic type checking judgments:

$\Gamma^+ \vdash \tau^+\ \textbf{type}$	`tok g t`	Check that τ is a type in Γ
$\Gamma^+ \vdash e^+ \Leftarrow \tau^+$	`ck g e t`	Check e against type τ
$\Gamma^+ \vdash e^+ \Rightarrow \tau^-$	`syn g e`	Infer a type τ for e
$\Gamma^+ \vdash \tau_1^+ \sqsubseteq \tau_2^+$	`sub g t1 t2`	Check τ_1 a subtype of τ_2

(Remember that a superscript $^+$ on a part of the judgment signifies that it is supplied as *input* when we read the judgment as an algorithm, while the superscript $^-$ means that the part is generated as *output*.) In the following rules, you should check for yourself that the modes are correct. The names of the functions that implement the algorithm in the SML code are also shown.

Type Validity First, we have the rules for the well-formedness of a type. In lecture we summarized this by saying that all the free variables of the type should be in the context. In the SML implementation of the language, a function `tok` ("type okay") performs this check. Since both type variables and term (value) variables are represented by de Bruijn indices, it's necessary to go through the type, looking each de Bruijn index up in the context and seeing that it corresponds to a type variable, not a value variable. There are also conditions as usual saying that whenever we put a variable in the context, it's not there already; I'll take those as understood without being written explicitly.

$$\frac{}{\Gamma, (\alpha\ \textbf{type}), \Gamma' \vdash \alpha\ \textbf{type}}$$

$$\frac{}{\Gamma \vdash \textbf{int type}} \qquad \frac{}{\Gamma \vdash \textbf{bool type}} \qquad \frac{}{\Gamma \vdash \textbf{float type}}$$

$$\frac{\Gamma \vdash \tau_1\ \textbf{type} \quad \Gamma \vdash \tau_2\ \textbf{type}}{\Gamma \vdash (\tau_1 \to \tau_2)\ \textbf{type}} \qquad \frac{\Gamma \vdash \tau\ \textbf{type}}{\Gamma \vdash (\tau\ \textbf{ref})\ \textbf{type}} \qquad \frac{\Gamma \vdash \tau\ \textbf{type}}{\Gamma \vdash (\tau\ \textbf{cont})\ \textbf{type}}$$

$$\frac{\Gamma \vdash \tau_1\ \textbf{type} \quad \ldots \quad \Gamma \vdash \tau_n\ \textbf{type}}{\Gamma \vdash (\tau_1 \times \cdots \times \tau_n)\ \textbf{type}} \qquad \frac{\Gamma \vdash \tau_1\ \textbf{type} \quad \ldots \quad \Gamma \vdash \tau_n\ \textbf{type}}{\Gamma \vdash (\tau_1 + \cdots + \tau_n)\ \textbf{type}}$$

$$\frac{\Gamma, (\alpha\ \textbf{type}) \vdash \tau\ \textbf{type}}{\Gamma \vdash (\forall \alpha.\, \tau)\ \textbf{type}} \qquad \frac{\Gamma, (\alpha\ \textbf{type}) \vdash \tau\ \textbf{type}}{\Gamma \vdash (\exists \alpha.\, \tau)\ \textbf{type}} \qquad \frac{\Gamma, (\alpha\ \textbf{type}) \vdash \tau\ \textbf{type}}{\Gamma \vdash (\mu \alpha.\, \tau)\ \textbf{type}}$$

Subtyping Next, we consider the subtyping rules, checked by the SML function `sub`. Generally, we think of the subtyping relation $\tau_1 \leq \tau_2$ as having transitivity, which wouldn't be mode correct (in checking that $\tau_1 \leq \tau_2$, we might have to guess a type σ and check $\tau_1 \leq \sigma$ and $\sigma \leq \tau_2$). Fortunately, we can write the rules in such a way that transitivity ends up being a *theorem*, rather than needing to be added as a rule. When we do subtyping this way, we write $\tau_1 \sqsubseteq \tau_2$ rather than $\tau_1 \leq \tau_2$ and call it *algorithmic subtyping.* We could also remove the reflexivity rule, but since it's okay as to mode correctness, it's fine to leave it in, too.

One more thing is that generally, in order to prove any theorems about subtyping, it's necessary to have the contexts hanging around too. (Otherwise, at various points in the proofs, it's hard to argue that the types involved are well-formed. What context should they be well-formed in?) So our algorithmic subtyping judgment will actually be written $\Gamma \vdash \tau_1 \sqsubseteq \tau_2$.

Without further ado, the subtyping rules:

$$\frac{}{\Gamma \vdash \tau \sqsubseteq \tau} \qquad \frac{}{\Gamma \vdash \textbf{int} \sqsubseteq \textbf{float}} \qquad \frac{\Gamma \vdash \sigma_1 \sqsubseteq \tau_1 \quad \Gamma \vdash \tau_2 \sqsubseteq \sigma_2}{\Gamma \vdash (\tau_1 \to \tau_2) \sqsubseteq (\sigma_1 \to \sigma_2)}$$

$$\frac{\Gamma \vdash \tau_1 \sqsubseteq \sigma_1 \quad \dots \quad \Gamma \vdash \tau_n \sqsubseteq \sigma_n}{\Gamma \vdash (\tau_1 \times \cdots \times \tau_n) \sqsubseteq (\sigma_1 \times \cdots \times \sigma_n)} \qquad \frac{\Gamma \vdash \tau_1 \sqsubseteq \sigma_1 \quad \dots \quad \Gamma \vdash \tau_n \sqsubseteq \sigma_n}{\Gamma \vdash (\tau_1 + \cdots + \tau_n) \sqsubseteq (\sigma_1 + \cdots + \sigma_n)}$$

$$\frac{\Gamma, (\alpha\ \textbf{type}) \vdash \tau \sqsubseteq \sigma}{\Gamma \vdash (\forall \alpha.\, \tau) \sqsubseteq (\forall \alpha.\, \sigma)} \qquad \frac{\Gamma, (\alpha\ \textbf{type}) \vdash \tau \sqsubseteq \sigma}{\Gamma \vdash (\exists \alpha.\, \tau) \sqsubseteq (\exists \alpha.\, \sigma)}$$

By adding the reflexivity rule, we've been able to get away without a bunch of boring axioms like **bool** $\sqsubseteq$ **bool** or $\alpha \sqsubseteq \alpha$.

You'll note that nothing was said about the type constructors $\tau\,$**ref**, $\tau\,$**cont**, or $\mu\alpha.\,\tau$. In the case of **ref**, neither covariant nor contravariant subtyping would work, for the reasons discussed in lecture. For **cont**, we could have a contravariant subtyping rule, but the associated coercion is tricky enough that adding the **cont** rule is left as an extra credit problem. Finally, we come to the recursive types $\mu\alpha.\,\tau$. On last fall's version of this assignment, the following UNSOUND rule was given:

$$¿ \quad \frac{\Gamma, (\alpha\ \textbf{type}) \vdash \tau \sqsubseteq \sigma}{\Gamma \vdash (\mu\alpha.\, \tau) \sqsubseteq (\mu\alpha.\, \sigma)} \quad ?$$

It's worth thinking about why this rule doesn't work, and what a correct rule might look like.

Q 3.1. [extra credit] (Somewhat tricky.) Show how to use the bogus rule above to write a program of type **int** that evaluates to 3.14159.

Checking Next we come to the rules for checking an expression against a known input type: $\Gamma \vdash e \Leftarrow \tau$. These are implemented by the SML function `ck`. It turns out that there are three general classes of rules which fall into this category. First of all, we have rules for constructing things of all the various types for which things can be explicitly constructed:

$$\frac{}{\Gamma \vdash n \Leftarrow \mathbf{int}} \qquad \frac{}{\Gamma \vdash f \Leftarrow \mathbf{float}}$$

$$\frac{}{\Gamma \vdash \mathbf{true} \Leftarrow \mathbf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} \Leftarrow \mathbf{bool}}$$

$$\frac{\Gamma, (f : \tau_1 \to \tau_2), (x : \tau_1) \vdash e \Leftarrow \tau_2}{\Gamma \vdash (\mathbf{fun}\, f(x).\, e) \Leftarrow (\tau_1 \to \tau_2)} \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (\mathbf{ref}\, e) \Leftarrow (\tau\, \mathbf{ref})}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \tau_1 \quad \ldots \quad \Gamma \vdash e_n \Leftarrow \tau_n}{\Gamma \vdash (e_1, \ldots, e_n) \Leftarrow (\tau_1 \times \cdots \times \tau_n)} \qquad \frac{\Gamma \vdash e \Leftarrow \tau_i}{\Gamma \vdash \mathbf{in}_i\, e \Leftarrow (\tau_1 + \cdots + \tau_n)}$$

$$\frac{\Gamma, (\alpha\, \mathbf{type}) \vdash e \Leftarrow \tau}{\Gamma \vdash (\Lambda\alpha.\, e) \Leftarrow (\forall\alpha.\, \tau)} \qquad \frac{\Gamma \vdash \tau\, \mathbf{type} \quad \Gamma \vdash e \Leftarrow \{\tau/\alpha\}\sigma}{\Gamma \vdash \mathbf{pack}(\tau, e) \Leftarrow (\exists\alpha.\, \sigma)}$$

$$\frac{\Gamma \vdash e \Leftarrow \{\mu\alpha.\, \tau/\alpha\}\tau}{\Gamma \vdash \mathbf{roll}\, e \Leftarrow (\mu\alpha.\, \tau)}$$

Notice how nicely we can treat **pack** and **roll**, for instance, because we assume that the **pack** or **roll** is located at a position in the term in which we already know what existential or recursive type the result is supposed to have. We also got rid of the big annotation on the injection $\mathbf{in}_i\, e$. You should verify to yourself that all these rules are mode correct.

In the next category we have various rules for deconstructing things, where the type being deconstructed isn't related to the type of the whole expression. Here there's mixing of the checking judgment (for the whole expression) and the synthesis judgment (for the part of the expression being deconstructed).

$$\frac{\Gamma \vdash e \Rightarrow \mathbf{bool} \quad \Gamma \vdash e_1 \Leftarrow \sigma \quad \Gamma \vdash e_2 \Leftarrow \sigma}{\Gamma \vdash (\mathbf{if}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2) \Leftarrow \sigma}$$

$$\frac{\Gamma \vdash e \Rightarrow (\tau_1 + \cdots + \tau_n) \quad \Gamma, (x_1 : \tau_1) \vdash e_1 \Leftarrow \sigma \quad \ldots \quad \Gamma, (x_n : \tau_n) \vdash e_n \Leftarrow \sigma}{\Gamma \vdash (\mathbf{case}\, e\, \mathbf{of}\, \mathbf{in}_1\, x_1 \Rightarrow e_1 \mid \ldots \mid \mathbf{in}_n\, x_n \Rightarrow e_n) \Leftarrow \sigma}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow (\exists\alpha.\, \tau) \quad \Gamma, (\alpha\, \mathbf{type}), (x : \tau) \vdash e_2 \Leftarrow \sigma}{\Gamma \vdash (\mathbf{unpack}(\alpha, x) = e_1\, \mathbf{in}\, e_2) \Leftarrow \sigma}$$

$$\frac{\Gamma \vdash e_2 \Rightarrow (\tau\, \mathbf{cont}) \quad \Gamma \vdash e_1 \Leftarrow \tau}{\Gamma \vdash (\mathbf{throw}\, e_1\, \mathbf{to}\, e_2) \Leftarrow \sigma}$$

In the third rule, we need the special condition that α isn't free in σ. The last rule is only mode correct if we put the premises in "backwards." This suggests that the syntax of **throw** might have been better the other way around. (Maybe **goto** e_2 **with** e_1?)

The third category is more of a catchall for rules that don't construct or deconstruct anything, really:

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau \quad \Gamma, (x:\tau) \vdash e_2 \Leftarrow \sigma}{\Gamma \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) \Leftarrow \sigma} \qquad \frac{\Gamma \vdash \tau\ \textbf{type} \quad \Gamma \vdash \{\tau/\alpha\}e \Leftarrow \sigma}{\Gamma \vdash (\textbf{let type}\ \alpha = \tau\ \textbf{in}\ e) \Leftarrow \sigma}$$

$$\frac{\Gamma, (x:\sigma\ \textbf{cont}) \vdash e \Leftarrow \sigma}{\Gamma \vdash (\textbf{letcc}\ x\ \textbf{in}\ e) \Leftarrow \sigma} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau \quad \Gamma \vdash e_2 \Leftarrow \sigma}{\Gamma \vdash (e_1; e_2) \Leftarrow \sigma}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \sigma \quad \Gamma \vdash e_2 \Leftarrow \sigma}{\Gamma \vdash (\textbf{try}\ e_1\ \textbf{ow}\ e_2) \Leftarrow \sigma} \qquad \frac{}{\Gamma \vdash \textbf{fail} \Leftarrow \sigma}$$

$$\frac{\Gamma \vdash e \Rightarrow \textbf{int}}{\Gamma \vdash \textbf{itof}\, e \Leftarrow \textbf{float}} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \quad \Gamma \vdash \tau \sqsubseteq \sigma}{\Gamma \vdash e \Leftarrow \sigma}$$

The rule for **itof** looks a bit strange. We could certainly give it different modes. (Actually, we could give it all four possible mode combinations, and it would still be mode correct). But since **itof** is going to be inserted in place of a use of subsumption (the last rule) it had better have exactly the same modes as subsumption, or strange things could happen. We want the result of the coercion elaboration to be well-typed without changing any of the modes around.

Synthesis Finally, we need to think about the rules for synthesizing a type for expressions for which this makes sense. We will be quite restrictive about this, only giving synthesis rules for constructs for which there was not also a checking rule. This seems restrictive, but in a realistically larger language, it becomes quite important. One reason is that if there's only one mode in which any particular construct can be handled, there's no non-determinism in the algorithm. All these rules are implemented by the SML function `syn`.

Most of the synthesis rules are about deconstructing something:

$$\frac{\Gamma \vdash e_1 \Rightarrow (\sigma \to \tau) \quad \Gamma \vdash e_2 \Leftarrow \sigma}{\Gamma \vdash (e_1\ e_2) \Rightarrow \tau} \qquad \frac{\Gamma \vdash e \Rightarrow (\tau_1 \times \cdots \times \tau_n)}{\Gamma \vdash e.\, i \Rightarrow \tau_i}$$

$$\frac{\Gamma \vdash e \Rightarrow (\tau\ \textbf{ref})}{\Gamma \vdash\ !e \Rightarrow \tau} \qquad \frac{\Gamma \vdash e_1 \Rightarrow (\tau\ \textbf{ref}) \quad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash (e_1 := e_2) \Rightarrow \tau}$$

$$\frac{\Gamma \vdash e \Rightarrow (\forall \alpha.\, \tau) \quad \Gamma \vdash \sigma\ \textbf{type}}{\Gamma \vdash e[\sigma] \Rightarrow \{\sigma/\alpha\}\tau} \qquad \frac{\Gamma \vdash e \Rightarrow (\mu\alpha.\, \tau)}{\Gamma \vdash \textbf{unroll}\, e \Rightarrow \{\mu\alpha.\, \tau/\alpha\}\tau}$$

The remaining synthesis rules handle variables, primitive operations such as $(+)$ and the like, and the type annotation construct:

$$\frac{(\mathit{opr} : \tau_1 \times \cdots \times \tau_n \to \sigma) \quad \Gamma \vdash e_1 \Leftarrow \tau_1 \quad \ldots \quad \Gamma \vdash e_n \Leftarrow \tau_n}{\Gamma \vdash \mathit{opr}(e_1, \ldots, e_n) \Rightarrow \sigma}$$

$$\frac{}{\Gamma, (x:\tau), \Gamma' \vdash x \Rightarrow \tau} \qquad \frac{\Gamma \vdash \tau\ \textbf{type} \quad \Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (e:\tau) \Rightarrow \tau}$$

The rule for variables makes sense because the modes for all of the judgments specify that the context Γ is *input.* (If you look back at all the rules that put variables in the context, you'll see that in each case the type is known.) Thus, when we come to a variable, we can find its type in the context and produce that type as output, so variables get a synthesis rule.

A somewhat subtle point about the bidirectional system is that it lets us get away with fewer checks on the well-formedness of types, as long as we think about the algorithm correctly. The general invariant is that we *assume* that any types mentioned in Γ or τ (if τ is an input) are already well-formed before the algorithm that decides the judgment runs. Conversely, when the algorithm finishes, if it produces a type as output, it is responsible for *guaranteeing* that the output type is well-formed. With this invariant, pretty much the only time a type needs to be checked for well-formedness is when it's something that was mentioned explicitly in the expression being checked (like in the syntax for ascriptions, or for applying a type function).

This completes the initial static semantics of our language. In a later section we will extend it to generate coercions to replace instances of the subsumption rule.

§ 4. De Bruijn Indices for Type Variables Before we get to the rules for generating coercions, I want to mention an interesting phenomenon that occurs when you use de Bruijn indices for type variables.

Looking at the rule for typing functions:

$$\frac{\Gamma, (f : \tau_1 \to \tau_2), (x : \tau_1) \vdash e \Leftarrow \tau_2}{\Gamma \vdash (\mathbf{fun}\ f(x).\, e) \Leftarrow (\tau_1 \to \tau_2)}$$

let's think about what it looks like in de Bruijn representation. To make this more concrete, let's pick a specific context and types:

$$\frac{(\alpha\,\mathbf{type}), (y : \alpha), (f : \alpha \to \alpha \times \alpha), (x : \alpha) \vdash (x, y) \Leftarrow (\alpha \times \alpha)}{(\alpha\,\mathbf{type}), (y : \alpha) \vdash (\mathbf{fun}\ f(x).\, (x, y)) \Leftarrow (\alpha \to \alpha \times \alpha)}$$

Now what should the de Bruijn version look like? This is tricky because there are type variables hanging around in the context, and in order to give them numbers, we have to decide what how many "nested binders" to count. We want to think of each item in the context as being a binder, the scope of which extends everywhere to its right, both within the context, and in the expression and type on the right side of the turnstile.

Following this rule, we arrive at:

$$\frac{(-\ \mathbf{type}), (- : \#1), (- : \#2 \to \#2 \times \#2), (- : \#3) \vdash (\#1, \#3) \Leftarrow (\#4 \times \#4)}{(-\ \mathbf{type}), (- : \#1) \vdash (\mathbf{fun}\ -(-).\, (\#1, \#3)) \Leftarrow (\#2 \to \#2 \times \#2)}$$

One way to check that this is right is to recall that we have the invariant that whenever the checking algorithm is invoked with $\Gamma \vdash e \Leftarrow \tau$, we should have $\Gamma \vdash \tau\ \mathbf{type}$. In this case, that means that we should have

$$(-\ \mathbf{type}), (- : \#1) \vdash (\#2 \to \#2 \times \#2)\ \mathbf{type}$$

in the conclusion, which indeed checks out, and

$$(-\ \mathbf{type}), (- : \#1), (- : \#2 \to \#2 \times \#2), (- : \#3) \vdash (\#4 \times \#4)\ \mathbf{type}$$

in the premise, which also checks out.

But this means that the *same type* can have a *different form* when it's carried from the conclusion into the premise, or when it moves into various places in the context. In fact, the type $(\#2 \to \#2 \times \#2)$ in the conclusion takes *three* different forms in the premise: it's used in the hypothesis for the type of the function name: $(- : \#2 \to \#2 \times \#2)$; part of it is used in the hypothesis for the type of the function argument: $(- : \#3)$; and part of it is used in the right-hand side of the premise as the type of the function's body: $(\#4 \times \#4)$.

All of these different forms arise by *shifting* the indices in the type up by different amounts. We want to summarize the outcome of this shifting process without getting drowned in numbers, so we'll use the notation $\tau^{(n)}$ for the result of taking a type τ in de Bruijn notation and shifting all its free indices by n (so $\#1$ becomes $\#(n+1)$, and so on).

With this notation, we can restate the general form of the function typing rule for de Bruijn indices as follows:

$$\frac{\Gamma, (f : \tau_1 \to \tau_2), (x : \tau_1^{(1)}) \vdash e \Leftarrow \tau_2^{(2)}}{\Gamma \vdash (\mathbf{fun}\ f(x).\, e) \Leftarrow (\tau_1 \to \tau_2)}$$

Here the names f and x are only mnemonics—they don't actually occur in the de Bruijn representation. Nothing in e itself needs to be shifted, because anything in e is already in the scope of Γ, f, and x in the conclusion, and is still in the scope of Γ, f, and x in the premise—the number of binders hasn't changed.

Certain laws help us reason about this shifting process. For instance, if $\Gamma \vdash \tau\ \mathbf{type}$, and Γ' has length n, then $\Gamma, \Gamma' \vdash \tau^{(n)}\ \mathbf{type}$. Also, whenever $\Gamma, (x : \tau), \Gamma'$ is well-formed, then $\Gamma \vdash \tau\ \mathbf{type}$.

There are two reasons I'm belaboring all this: first of all, the code you'll be working with has all of this shifting in it, and I don't want you to be confused. Second, this is tricky stuff. The sample "solution" handed out last fall for this assignment got all this totally wrong. I think the only reason the problems with it weren't discovered was that, amusingly, all the various de Bruijn index bugs in it had mostly canceled each other out.

So although initially this assignment was to have had you writing part of the code for the bidirectional checker yourselves, it seemed cruel and unusual punishment to make you do it when not even the teaching staff got it right last fall.

§ 5. Generating Coercions Now we've almost come—finally!—to the actual questions.

The initial code for this assignment is able to typecheck certain programs that it can't run. For example, using `Top.itype()` you can verify that

$$(\mathbf{fun}\ f(x).\, \mathbf{sqrt}(x +.\ 3.14159) : \mathbf{float} \to \mathbf{float})(1 : \mathbf{int}) : \mathbf{float}$$

typechecks, but if you try to run it with `Top.ieval_noelab()`, the machine gets stuck. The reason is that the type system allows subsumption to silently convert an **int** to a **float**, but the operational semantics can't do that. (Remember that **int**s and **float**s might need to be in different registers, or something.)

What we want is for all the uses of subsumption to be *elaborated* into explicit coercions. If we're just subsuming **int** to **float**, the corresponding coercion will be the primitive operation **itof**. With more complicated uses of subsumption, though, the coercions will be more complex.

Stubs for the code to do this are already given in `typing.sml`; if you fill them in, then the `Top.ieval()` function will use them to elaborate terms before sending them to the evaluator, and thus the function above should run properly rather than getting stuck.

The basic idea is that you'll instrument the static semantics above with extra outputs giving the elaborated form of the expressions being checked:

$\Gamma^+ \vdash e^+ \Leftarrow \tau^+ \rightsquigarrow e'^-$	`elck g e t` Check e against type τ returning e'
$\Gamma^+ \vdash e^+ \Rightarrow \tau^- \rightsquigarrow e'^-$	`elsyn g e` Infer a type τ for e and return e'
$\Gamma^+ \vdash \tau_1^+ \sqsubseteq \tau_2^+ / e^+ \rightsquigarrow e'^-$	`elsub g t1 t2 e` Check τ_1 a subtype of τ_2 and convert e to e'

Thus, there will be a written part, where you give the rules for these judgments, and a programming part, where you implement them.

To get you started, here is the rule, alluded to above, for converting an **int** to a **float**:

$$\frac{}{\Gamma \vdash \textbf{int} \sqsubseteq \textbf{float}/e \rightsquigarrow (\textbf{itof}\, e)}$$

And here is the subsumption rule instrumented with elaborations:

$$\frac{\Gamma \vdash e \Rightarrow \tau_1 \rightsquigarrow e' \quad \Gamma \vdash \tau_1 \sqsubseteq \tau_2/e' \rightsquigarrow e''}{\Gamma \vdash e \Leftarrow \tau_2 \rightsquigarrow e''}$$

The mode correctness is now a bit tricky. First, in the conclusion, we get Γ and e and τ_2 as inputs. Next, by invoking the algorithm for the first premise, we get τ_1 and e' as outputs. Then, invoking the algorithm for the second premise, we get e'' as output. Finally, we return e'' in the conclusion as output to our caller.

As another example, here's the rule for applications:

$$\frac{\Gamma \vdash e_1 \Rightarrow (\sigma \to \tau) \rightsquigarrow e_1' \quad \Gamma \vdash e_2 \Leftarrow \sigma \rightsquigarrow e_2'}{\Gamma \vdash (e_1\, e_2) \Rightarrow \tau \rightsquigarrow (e_1'\, e_2')}$$

The rule for **let type** we'll be using is the following:

$$\frac{\Gamma \vdash \tau\, \textbf{type} \quad \Gamma \vdash \{\tau/\alpha\}e \Leftarrow \sigma \rightsquigarrow e'}{\Gamma \vdash (\textbf{let type}\ \alpha = \tau\ \textbf{in}\ e) \Leftarrow \sigma \rightsquigarrow e'}$$

This is the only elaboration rule where some kind of "evaluation" happens; that is, the **let type** is no longer present in the result e' of the elaboration. In general, it's hard to see how e' could be put back into the form **let type** $\alpha = \tau$ **in** $\ldots$, at least not in a uniquely determined way.

And a final example, the rule for annotations:

$$\frac{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow e'}{\Gamma \vdash (e : \tau) \Rightarrow \tau \rightsquigarrow (e' : \tau)}$$

(The annotation is still there in the output; otherwise, the output wouldn't typecheck.)

In general, we'd like the following theorems to be true (though you don't have to write up the proofs).

1. If $\Gamma \vdash e \Leftarrow \tau$, then $\Gamma \vdash e \Leftarrow \tau \rightsquigarrow e'$ for some e', and furthermore $\Gamma \vdash e' \Leftarrow \tau$ *without* any subsumptions (other than reflexivity).

2. If $\Gamma \vdash e \Rightarrow \tau$, then $\Gamma \vdash e \Rightarrow \tau \rightsquigarrow e'$ for some e', and furthermore $\Gamma \vdash e' \Rightarrow \tau$ without subsumptions other than reflexivity.

3. If $\Gamma \vdash \tau_1 \sqsubseteq \tau_2$, and $\Gamma \vdash e \Rightarrow \tau_1$, then $\Gamma \vdash \tau_1 \sqsubseteq \tau_2/e \rightsquigarrow e'$ for some e', and furthermore $\Gamma \vdash e' \Leftarrow \tau_2$ without any non-trivial subsumptions (assuming e doesn't use any non-trivial subsumptions).

Take particular note of the modes in part 3. They ensure that the coercion from e to e' fits into the same slot that the original use of the subsumption rule did, because the modes are the same as for the subsumption rule.

Conversely, we don't want the system instrumented with elaborations to type any *more* programs than the original system:

1. If $\Gamma \vdash e \Leftarrow \tau \rightsquigarrow e'$, then $\Gamma \vdash e \Leftarrow \tau$.

2. If $\Gamma \vdash e \Rightarrow \tau \rightsquigarrow e'$, then $\Gamma \vdash e \Rightarrow \tau$.

3. If $\Gamma \vdash \tau_1 \sqsubseteq \tau_2/e \rightsquigarrow e'$, then $\Gamma \vdash \tau_1 \sqsubseteq \tau_2$.

Finally, although we won't make the notion precise here, the original expression and the elaborated one should "do the same thing" in the operational semantics. You shouldn't elaborate **true** into **false**, for instance. (The difficulty in making this precise is that part of the point is that the original term and the elaborated term don't do *exactly* the same thing, because one gets stuck and the other doesn't!)

Q 5.1. [45 pts] Write up instrumented versions of all the typing rules (the named-form rules, not the de Bruijn ones) given in Section 3 of the assignment. This includes the rules for $\Gamma \vdash e \Rightarrow \tau \rightsquigarrow e'$, for $\Gamma \vdash e \Leftarrow \tau \rightsquigarrow e'$, and for $\Gamma \vdash \tau_1 \sqsubseteq \tau_2/e \rightsquigarrow e'$. The theorems above should hold for your rules, but you don't have to prove them. Turn your work in in lecture, or email it to me by 11:59 p.m. (If you do it by hand, you can always scan it and email me the scans.)

The final part of the assignment is to implement your elaboration rules. For this, you should fill in the stubs of `elsyn`, `elck`, and `elsub` in the file `typing.sml`. The code for `syn`, `ck`, and `sub` should be your model, except that you will need to add extra outputs (and an input, for `elsub`) to return the result of the elaboration.

You should not need to do any de Bruijn shifting, or call the functions for substitution, on the elaborated terms that you construct (although the part that you inherit from `ck` will still have to).

Q 5.2. [45 pts] Write code for the `elsyn`, `elck`, and `elsub` functions in `typing.sml`. Turn your modified file in to the handin directory in AFS.

Finally, you should check that `Top.ielab()`, which calls your elaboration functions, returns sensible results, and that `Top.ieval()`, which elaborates before evaluating, doesn't get stuck on some examples you cook up for which the unelaborated version `Top.ieval_noelab()` does.

Q 5.3. [10 pts] Put your test examples in a file `testelab.mml`, and hand it in to your AFS handin directory.