Topic 1. Type checking. By careful inspection of the type judgement derivation rules, it is easy to see that the analysis is decidable. In other words, you can define an algorithm for type checking, which we do via the pseudocode definition of TC as follows; TC(\(\Gamma, e\)) takes a free variable type binding environment \(\Gamma\) and expression \(e\) as arguments, and returns a type \(\tau\) as a result, and is defined via case analysis on the expression forms of TD:

\[
\begin{align*}
TC(\Gamma, n) &= \text{Int} \\
TC(\Gamma, b) &= \text{Bool} \\
TC(\Gamma, x) &= \Gamma(x) \\
TC(\Gamma, e_1 + e_2) &= \text{if } TC(\Gamma, e_1) = \text{Int} \\
&\quad \text{and } TC(\Gamma, e_2) = \text{Int} \\
&\quad \text{then } \text{Int} \\
&\quad \text{else raise TypeMismatch} \\
TC(\Gamma, e_1 e_2) &= \text{if } TC(\Gamma, e_1) = \tau' \rightarrow \tau \\
&\quad \text{and } TC(\Gamma, e_2) = \tau'' \\
&\quad \text{and } \tau' = \tau'' \\
&\quad \text{then } \tau \\
&\quad \text{else raise TypeMismatch} \\
TC(\Gamma, \text{Function } (x : \tau_1 \rightarrow e)) &= \text{let } \tau_2 = TC(\Gamma; x : \tau_1, e) \\
&\quad \text{in } \tau_1 \rightarrow \tau_2 \\
\end{align*}
\]

We can be very precise about the desired behavior of TC:

**Theorem 1.1** \(TC(\Gamma, e) = \tau\) if and only if \(\Gamma \vdash e : \tau\).

Topic 2. OCaml implementation. To implement the TD type checking analysis in OCaml, we define an abstract type syntax:

\[
\text{type dtype} = \text{Intt} \mid \text{Boolt} \mid \text{Arrow of dtype * dtype}
\]

A transformation from TD concrete types to abstract types is easily defined:

\[
\begin{align*}
\llbracket\text{Int}\rrbracket &= \text{Intt} \\
\llbracket\text{Bool}\rrbracket &= \text{Boolt} \\
\llbracket\tau_1 \rightarrow \tau_2\rrbracket &= \text{Arrow}(\llbracket\tau_1\rrbracket, \llbracket\tau_2\rrbracket)
\end{align*}
\]

Imagining an encoding of type environments as lists \(l : (\text{ident} * \text{dtype})\) list, we extend our transformation to environments as follows:

\[
\llbracket(x_1 : \tau_1; \ldots; x_n : \tau_n)\rrbracket = [(\text{Ident("x1")}, \llbracket\tau_1\rrbracket), \ldots, (\text{Ident("xn")}, \llbracket\tau_n\rrbracket)]
\]
And define a lookup operation such that for all $x$, we have:

$$\text{lookup } \text{Ident}("x") \downarrow \Gamma(x)$$

Then, simply transcribe the TC algorithm into OCaml—this is homework assignment 103, where you are to define the function $\text{typecheck}$. Since this is essentially a transcription of the derivation rules, it is trivial to establish correctness of the OCaml $\text{typecheck}$ algorithm:

**Theorem 1.2** The judgement $\Gamma \vdash e : \tau$ is derivable if and only if $\text{typecheck } \Gamma(e) \downarrow \tau$.

**Topic 3. Incompleteness of type analysis.** Type disciplines, in order to be tractable or even decidable, cut corners and rule out some safe expressions, such as:

$$1 + (\text{If True Then 10 Else False})$$

An interesting class of expressions that are ruled out by type checking are functions that force variables to self-apply, e.g. (in D):

$$(\text{Function } x \to xx)$$

The problem is, picking a type $\tau$ such that the function will type check in TD as $\text{Function } (x : \tau) \to xx$; it can’t be done (think about it...). This turns out to be quite significant, since without an explicit recursive binding mechanism such as $\text{let rec}$ or $\text{Fix}$, you need self-application of variables to encode recursion. In other words, TD without $\text{Fix}$ is not Turing complete! (It also turns out to be strongly normalizing).

**Topic 4. Variable binding with Let and Fix.** We now extend TD with the syntax, semantics, and typing rules for variable declarations and recursively bound function definitions. The concrete syntax of expressions and values is extended as follows:

$$v ::= \cdots | (\text{Fix } z.(x : \tau) : \tau \to e) \quad \text{values}$$

$$e ::= \cdots | \text{Let } x = e \text{ In } e \quad \text{expressions}$$

The semantics of TD are extended by adapting the Let semantics of D, and modifying the $\text{APPLFIX}$ rule to ignore type information:

$$\text{APPLFIX}$$

$$e_1 \Rightarrow (\text{Fix } z.(x : \tau_1) : \tau_2 \to e) \quad e_2 \Rightarrow \nu' \quad e[(\text{Fix } z.(x : \tau_1) : \tau_2 \to e)/z][\nu'/x] \Rightarrow \nu$$

$$e_1 e_2 \Rightarrow \nu$$

Typing is extended as follows. Note that there is no need to declare the type $x$ in expressions $\text{Let } x = e_1 \text{ In } e_2$, since the type of the given expression $e_1$ determines the type of $x$:

$$\text{LET}$$

$$\Gamma \vdash e_1 : \tau' \quad \Gamma ; x : \tau' \vdash e_2 : \tau \quad \Gamma \vdash \text{Let } x = e_1 \text{ In } e_2 : \tau$$

$$\text{FIX}$$

$$\Gamma ; z : \tau_1 \Rightarrow \tau_2 ; x : \tau_1 \vdash e : \tau_2$$

$$\Gamma \vdash (\text{Fix } z.(x : \tau_1) : \tau_2 \to e) : \tau_1 \Rightarrow \tau_2$$

And yes, we now have Turing completeness (but not strong normalization) in TD. We extend the abstract syntax of expressions as follows, while translation of the type checking and evaluation rules are left as an exercise in homework 103:

$$\text{type expr} =$$

$$\cdots$$

$$| \text{Fix of ident * ident * dtype * dtype * expr}$$

$$| \text{Let of ident * expr * expr}$$

The concrete-to-abstract encoding:

$$\Gamma \vdash \text{Let } x = e_1 \text{ In } e_2$$

$$\Gamma \vdash (\text{Fix } z.(x : \tau_1) : \tau_2 \to e)$$

$$\Gamma \vdash \text{Let } \text{Ident}"x"$, $\Gamma_1$, $\Gamma_2]$$

$$\Gamma \vdash \text{Fix } \text{Ident}"x"$, $\Gamma_1$, $\Gamma_2$, $\Gamma_3$$
**Topic 5. Properties of type checking.** Given the definitions of evaluation and type checking, we can now revisit two running themes in this course—although previously they’ve been stated in vague terms, now we can be mathematically precise about our meaning. Firstly, recall our running intuition about types— that the static type of an expression predicts the class of values to which the expression will evaluate. This can be framed as the following *type preservation* theorem:

**Theorem 1.3 (Type preservation)** If \( \Gamma \vdash e : \tau \) and \( e \Rightarrow v \) then \( \Gamma \vdash v : \tau \).

The theorem follows by structural induction on the derivation of \( e \Rightarrow v \). On the basis of type preservation, it is also possible to prove a *type safety* result, which demonstrates that type analysis rules out unsafe expressions:

**Theorem 1.4 (Type safety)** If \( e : \tau \) then it is not the case that \( e \Rightarrow \text{fail} \).