THE SIMPLY TYPED LAMBDA CALCULUS
Type checking is usually a single static phase that happens before any evaluation.

- a term "typechecks" when it has a valid type
- we don't want to evaluate any terms that don't typecheck.
The Simply Typed Lambda Calculus

- implement \texttt{typecheck :: Expr \rightarrow Ty}
  - where the target \texttt{Ty} is another datatype representing your types

- we write type-checking rules to define the types of terms
  - similar format as our evaluation rules
A Simple Starting Point

Here's the core simply typed lambda calculus, extended with bools and numbers.

\[
t ::= \lambda x:T.t \mid (t \ t) \mid x \\
    \mid \text{true} \mid \text{false} \mid \text{if } t \ t \ t \\
    \mid <\text{integers}> \mid t + t \mid t - t \mid t \cdot t \\
    \mid t < t \mid t > t
\]

\[
T ::= T \rightarrow T \mid \text{TyInt} \mid \text{TyBool}
\]

\[
v ::= \lambda x:T.t \mid \text{true} \mid \text{false} \mid <\text{integers}>
\]

"is this term well-typed?" is just as much work as "what is this term's type?"
The simply-typed lambda calculus ($\lambda\rightarrow$)

• We introduce **types** to the lambda calculus.
  • Each term in the language has a type
  • types could be implicit or explicit (our choice here)

• **type ascription operator, :**
  • true : Bool
  • 5 : Int
  • (Pair true 5) : TyPair Bool Int

see STLC.hs
Notes on $\lambda$→

The core simply typed lambda calculus is **strongly normalizing**

- no unbounded calculation – always halts!

- we lost the ability to do any recursion by giving terms types (why? try writing a type for the $\omega$- and ys-combinators.)

- but we can regain recursion by extending the language as before.
Maintaining an environment

typechecking
• navigates subterms to determine types
• but, no substitution occurs (we're not evaluating)

handling variables
• store their types when introduced
• at what type was it introduced by a lambda?
• all surrounding lambdas' variables' types must be tracked
Maintaining an environment

How can we store this information?

• We can keep a list of pairs, \([(\text{String}, \text{Ty})]\)
  • look up variables' types

• this "environment" is called Gamma (\(\Gamma\)).
Basic typing rules

- **Ty-var**
  \[ (x, T) \in \Gamma \]
  \[ \Gamma \vdash x : T \]

- **Ty-\(\lambda\)**
  \[ (\Gamma, (x, T_d)) \vdash t_r : T_r \]
  \[ \Gamma \vdash (\lambda x: T_d . t_r) : T_d \rightarrow T_r \]

- **Ty-app**
  \[ \Gamma \vdash t_1 : T_d \rightarrow T_r \]
  \[ \Gamma \vdash t_2 : T_d \]
  \[ \Gamma \vdash (t_1 \ t_2) : T_r \]

If \(\Gamma\) has \((x, T)\) added to it, then \(\Gamma\) can look up variable \(x\) and find its type is \(T\).

If we extend \(\Gamma\) with \((x, T)\) and that finds \(t_r\)'s type to be \(T_r\), then the original \(\Gamma\) derives the type of \((\lambda x: T_d . T_r)\) to be \(T_d \rightarrow T_r\).

If \(\Gamma\) finds \(t_1 : T_d \rightarrow T_r\) and \(\Gamma\) finds \(t_2 : T_d\), then \(\Gamma\) derives the overall type of \((t_1 \ t_2)\) to be \(T_r\).
Typing rules: Booleans

- **Ty-true**
  \[ \Gamma \vdash \text{true} : \text{TyBool} \]

- **Ty-false**
  \[ \Gamma \vdash \text{false} : \text{TyBool} \]

- **Ty-if**
  \[ \Gamma \vdash \text{a} : \text{TyBool}, \Gamma \vdash \text{b} : \text{T}, \Gamma \vdash \text{c} : \text{T} \]
  \[ \Gamma \vdash \text{if a b c} : \text{T} \]

Without an environment, we know the type of true is TyBool.

Same for false.

If \( \Gamma \) derives \( \text{a} \)'s type as TyBool, and \( \Gamma \) derives that both \( \text{b} \) and \( \text{c} \) have some type \( \text{T} \), then \( \Gamma \) derives the overall type of \( \text{if a b c} \) as \( \text{T} \).
Typing rules: Numbers

- **Ty-int**
  
  \[
  \Gamma \vdash <#> : \text{TyInt}
  \]
  
  With no \( \Gamma \) needed, we know any literal integer is of type \( \text{TyInt} \).

- **Ty-add**
  
  \[
  \Gamma \vdash t_1 : \text{TyInt} \quad \Gamma \vdash t_2 : \text{TyInt} \\
  \Gamma \vdash (t_1 + t_2) : \text{TyInt}
  \]
  
  If \( \Gamma \) derives that \( t_1 \) and \( t_2 \) are both of type \( \text{TyInt} \), then \( \Gamma \) derives \( (t_1 + t_2) \) to be of type \( \text{TyInt} \).

- **Ty-GT**
  
  \[
  \Gamma \vdash t_1 : \text{TyInt}, \quad \Gamma \vdash t_2 : \text{TyInt} \\
  \Gamma \vdash t_1 > t_2 : \text{TyBool}
  \]
  
  If \( \Gamma \) derives \( t_1 \) and \( t_2 \) are both of type \( \text{TyInt} \), then \( \Gamma \) derives that \( t_1 > t_2 \) is of type \( \text{TyBool} \).

\( \text{Ty-sub, Ty-mul, TyLT: similar to Ty-add and Ty-GT} \)
Using Typing Rules – Proof Trees

- We don't simplify, so our continual reduce-with-justification doesn't work directly as it did with evaluation rules.
- We write a **proof tree** from bottom up to show the claim of the bottom-most line.
- Each level uses a typing rule with actual terms plugged in.
- We can label which *rule* was used, but it's always the one applicable rule.

```
show that:  \((\lambda x:\text{TyInt}.x+1) \ 3\)  \ :  \text{TyInt}

\[
\begin{array}{c}
(x,\text{TyInt}) \vdash x:\text{TyInt}  & \vdash 1: \text{TyInt} \\
(x,\text{TyInt}) \vdash x+1: \text{TyInt}  & \vdash (\lambda x:\text{TyInt}.x+1) : \text{TyInt} \rightarrow \text{TyInt} \\
\vdash (\lambda x:\text{TyInt}.x+1) \ 3\)  \ :  \text{TyInt}
\end{array}
\]
```
Using Typing Rules – Proof Trees

show that: \(((\lambda x: \text{TyInt}.x+1) \ 3) : \text{TyInt}\)

\[
\begin{align*}
\frac{}{\vdash (x, \text{TyInt}) T x : \text{TyInt}} & \quad \frac{}{\vdash 1 : \text{TyInt}} \\
\frac{\vdash (x, \text{TyInt}) T x : \text{TyInt}}{\vdash (x, \text{TyInt}) T (x+1) : \text{TyInt}} & \quad \frac{}{\vdash 3 : \text{TyInt}} \\
\frac{}{\vdash (\lambda x: \text{TyInt}.x+1) : \text{TyInt} \rightarrow \text{TyInt}} & \quad \frac{\vdash (\lambda x: \text{TyInt}.x+1) : \text{TyInt} \rightarrow \text{TyInt}}{\vdash ((\lambda x: \text{TyInt}.x+1) \ 3) : \text{TyInt}}
\end{align*}
\]
Proof Tree Examples

show that: \( \text{if (1<2) 3 4 : TyInt} \)

\[
\begin{align*}
\vdash 1: \text{TyInt} & \quad \vdash 2: \text{TyInt} & \quad \vdash (1<2): \text{TyBool} & \quad \vdash 3: \text{TyInt} & \quad \vdash 4: \text{TyInt} \\
\vdash \text{if (1<2) 3 4 : TyInt} & 
\end{align*}
\]
Proof Tree Examples

try to show that:  \((3+4) \times \text{true} : \text{TyInt}\)

\[
\begin{align*}
\vdash 3 : \text{TyInt} & \quad \vdash 4 : \text{TyInt} \\
\phantom{\vdash} & \quad \phantom{\vdash} \\
\vdash (3+4) : \text{TyInt} & \quad \vdash \text{true} : \text{TyBool} \\
\vdash (3+4) \times \text{true} : \text{TyInt}
\end{align*}
\]

does not typecheck:

- \(\text{Ty-mul requires } t_2 \text{ to be } \text{TyInt, but } \text{true} : \text{TyBool}\)
- \((\text{Ty-mul's } t_1 \text{ term did typecheck, by the way})\)
Proof Tree Examples

show that:  \((\lambda x: TyBool. if \; x \; 3 \; 4) \; ((\lambda a: TyInt. a<5) \; 9)) \; : \; TyInt

\[
\begin{array}{c}
(x, TyBool) \vdash x : TyBool \\
\vdash 3 : TyInt \\
\vdash 4 : TyInt
\end{array}
\begin{array}{c}
(a, TyInt) \vdash a : TyInt \\
\vdash 5 : TyInt
\end{array}
\begin{array}{c}
(a, TyInt) \vdash (a<5) : TyBool
\end{array}
\begin{array}{c}
(x, TyBool) \vdash (if \; x \; 3 \; 4) : TyInt
\end{array}
\begin{array}{c}
\vdash (\lambda a : TyInt.a<5) : TyInt \rightarrow TyBool \\
\vdash 9 : TyInt
\end{array}
\begin{array}{c}
\vdash (\lambda x : TyBool.if \; x \; 3 \; 4) : TyBool \rightarrow TyInt
\end{array}
\begin{array}{c}
\vdash ((\lambda a : TyInt.a<5) \; 9) : TyBool
\end{array}
\begin{array}{c}
\vdash ((\lambda x : TyBool.if \; x \; 3 \; 4) \; ((\lambda a : TyInt.a<5) \; 9)) \; : \; TyInt
\end{array}
\]
Practice Problems – Proof Trees

Write full proof trees to find the type for each expression:

- \((\lambda x:\text{TyInt}.x) ((\lambda n:\text{TyInt}.n) 10))\)
- \((\lambda x:\text{TyInt}\text{.if true x (x+1)}) 5)\)
- \((\lambda f:\text{TyInt}\rightarrow\text{TyInt}. \lambda x:\text{TyInt}.f x)\)
- \(((\lambda x:\text{TyInt}. \lambda y:\text{TyInt}.x+y) 5) 8)\)

Explain why the following do not have proof trees (answer by discussing which subterms' needed types don't comply with the example's needs)

- \((3 5)\)
- \(((\lambda x:\text{TyInt}.x+1) \text{ true})\)
practice problems - encodings

Now that we have our language, let’s use it.

Encode these:
• and :: $B \rightarrow B \rightarrow B$
• or :: $B \rightarrow B \rightarrow B$
• ge:: $Z \rightarrow Z \rightarrow B$
• inc :: $Z \rightarrow Z$
• max3 :: $Z \rightarrow Z \rightarrow Z \rightarrow Z$

greater or equal than
Extending $\lambda$→

• Terms/values extended as before
  → often need explicit types, e.g. $\lambda x: T\cdot t$

• Often add one new type to $T ::= \text{for each extension}$

• Each term probably gets exactly one typing rule
  → no substitution, so easier than evaluation rules!
  → must maintain $\Gamma$, so trickier than evaluation rules!
  → usually invoke (:) on all subterms, used to claim overall term's type
Practice: Pairs Extension

Language additions:
\[
t ::= \ldots \mid \text{pair } t \ t \mid \text{fst } t \mid \text{snd } t
\]
\[
v ::= \ldots \mid \text{pair } t \ t
\]
\[
T ::= \ldots \mid \text{TyPair } T \ T
\]

Evaluation Rules:
(same as before)

Typing Rules:
(one per new term)

Ty-Pair: 
\[
\frac{\Gamma \vdash t_1: T_1, \Gamma \vdash t_2: T_2}{\Gamma \vdash \text{pair } t_1 t_2 : \text{TyPair } T_1 T_2}
\]

Ty-Fst: 
\[
\frac{\Gamma \vdash t: \text{TyPair } T_1 T_2}{\Gamma \vdash \text{fst } t : T_1}
\]

Ty-Snd: 
\[
\frac{\Gamma \vdash t: \text{TyPair } T_1 T_2}{\Gamma \vdash \text{snd } t : T_2}
\]
practice problems – Pairs + Proof Trees

Draw proof trees to find these expressions' types:

• $\text{fst (pair 3 true)}$
• $\lambda p : \text{TyPair TyInt TyBool}. \text{if (snd p) (fst p) 0}$
Practice: Recursion Extension

We again add Fix to the language, only now it involves typed lambdas.

**Language addition:**

\[ t ::= \ldots \mid \text{fix } t \]

**Evaluation Rule:**

(same as before, with type ascription)

\[
\text{E-Fix: } \quad \text{fix } (\lambda x: T . t) \rightarrow t [x \mapsto (\text{fix } (\lambda x: T . t))] 
\]

**Typing Rule:**

(one per new term)

\[
\text{Ty-Fix: } \quad \Gamma \vdash t : T \rightarrow T \\
\Gamma \vdash \text{fix } t : T
\]
practice problems - Recursion

Draw proof trees to find these expressions' types:

- \text{fix } (\lambda \text{self}: \mathbb{Z} \rightarrow \mathbb{Z}. \lambda n: \mathbb{Z}. \text{if } (n < 2) \ 1 \ (n \times (\text{self} \ (n-1))))
  
  Just sketch the bottom two levels to see Ty-Fix in use.

Encode these no-lists-involved things:

- factorial (shown above)
- fibonacci
**Practice: Lists Extension** (p.1/2)

**Language additions:**

\[

t ::= \ldots \mid \text{nil } T \mid \text{cons } t \ t \mid \text{head } t \mid \text{tail } t \mid \text{isnil } t
\]

\[
T ::= \ldots \mid \text{TyList } T
\]

\[
v ::= \ldots \mid \text{nil } T \mid \text{cons } t \ t
\]

**Evaluation Rules:**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| **E-head:**           | \[\begin{array}{c}
t \to t' \\
\hline
\text{head } t \to \text{head } t'
\end{array}\]                                    |
| **E-head-cons:**      | head(cons \( t_1 \ t_2 \)) \to t_1               |
| **E-tail:**           | \[\begin{array}{c}
t \to t' \\
\hline
\text{tail } t \to \text{tail } t'
\end{array}\]                                    |
| **E-tail-cons:**      | tail(cons \( t_1 \ t_2 \)) \to t_2              |
| **E-isnil:**          | \[\begin{array}{c}
t \to t' \\
\hline
\text{isnil } t \to \text{isnil } t'
\end{array}\]                                    |
| **E-isnil-true:**     | isnil (nil \( T \)) \to \text{true}             |
| **E-isnil-false:**    | isnil (cons \( t_1 \ t_2 \)) \to \text{false}   |
Practice: Lists Extension (p.2/2)

Language additions:
\[ t ::= \ldots \mid \text{nil} \ T \mid \text{cons} \ t \ t \mid \text{head} \ t \mid \text{tail} \ t \mid \text{isnil} \ t \]
\[ T ::= \ldots \mid \text{TyList} \ T \]
\[ v ::= \ldots \mid \text{nil} \ T \mid \text{cons} \ t \ t \]

Evaluation Rules:
(same as before)

Typing Rules: (one per new term)

- **Ty-nil**: \( \Gamma \vdash \text{nil} \ T : \text{TyList} \ T \)
- **Ty-cons**: \( \Gamma \vdash t_1 : T, \Gamma \vdash t_2 : \text{TyList} \ T \quad \Gamma \vdash \text{cons} \ t_1 \ t_2 : \text{TyList} \ T \)
- **Ty-head**: \( \Gamma \vdash t : \text{TyList} \ T \quad \Gamma \vdash \text{head} \ t : T \)
- **Ty-tail**: \( \Gamma \vdash t : \text{TyList} \ T \quad \Gamma \vdash \text{tail} \ t : \text{TyList} \ T \)
- **Ty-isnil**: \( \Gamma \vdash t : \text{TyList} \ T \quad \Gamma \vdash \text{isnil} \ t : \text{TyBool} \)
practice problems - Lists

Draw proof trees to find these expressions' types:

- head (if true (cons 5 (nil TyInt)) (nil TyInt)))
- head (tail (cons 10 (cons 12 (cons 13 (nil TyInt)))))

Encode these:

- length:: [Z] → Z
- map:: (Z→Z) → [Z] → [Z]
- nth :: [Z] → Z → Z. \textit{gets the nth item in the list} (or zero).
Thoughts

- With no extensions, $\lambda\rightarrow$ (with only $t ::= x | \lambda x : T . t | (t \ t)$) is degenerate (it has no values). Why?

- evaluation should *preserve types* – a term's value should not change types due to further evaluation.
  $\rightarrow$ true for $\lambda\rightarrow$ as presented

- erasure: after typechecking, we can erase all types in $\lambda\rightarrow$ and evaluation is unaffected.
  - Java's Generics were added this way!
  - "unerasing" is the process of inferring types
Curry-Howard Correspondence

- Strikingly similar features shared between logic and type theory.
- continues through many more complex features of type theory!

**Propositions as Types analogy**

<table>
<thead>
<tr>
<th>Logic</th>
<th>concept</th>
<th>Type Theory</th>
<th>concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposition</td>
<td>a statement (may be T/F)</td>
<td>types</td>
<td></td>
</tr>
<tr>
<td>$P \implies Q$</td>
<td>given proof $P$, make proof of $Q$</td>
<td>type $P \rightarrow Q$</td>
<td>function from $P$ to $Q$</td>
</tr>
<tr>
<td>$P \land Q$</td>
<td>stmt that $P$ and $Q$ are true</td>
<td>type $P \times Q$</td>
<td>product type (e.g. tuple)</td>
</tr>
<tr>
<td>$P \lor Q$</td>
<td>stmt that $P$ or $Q$ is true</td>
<td>type $P + Q$</td>
<td>union type (e.g. Either a b)</td>
</tr>
<tr>
<td>proof of $P$</td>
<td>way to show truth of $P$</td>
<td>term of type $P$</td>
<td>way to construct value</td>
</tr>
<tr>
<td>$P$ is provable</td>
<td>claim: $P$ is true</td>
<td>type $P$ inhabited</td>
<td>claim: elt of $P$ exists</td>
</tr>
</tbody>
</table>