THE SIMPLY TYPED LAMBDA CALCULUS

The Simply Typed Lambda Calculus

We will enhance our untyped lambda calculus **with types.**

Type checking is usually a single static phase that happens **before** any evaluation.

- a term "***typechecks***" (passes type-checking analysis) when it has a valid type
- we don't want to evaluate any terms that don't typecheck.

The Simply Typed Lambda Calculus

- implement `typecheck :: Tm -> Ty`
  - where the target `Ty` is another datatype representing your types
- we write type-checking rules to define the types of terms
  - similar format as our evaluation rules. Instead of $\longrightarrow$, we define **:**

A Simple Starting Point

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

```
t ::= λx:T.t | (t t) | x
    | true | false | if t t t
    | <integers> | t + t | t - t | t * t
    | t < t  |  t > t

T ::= T⟶T  |  ℤ  |  𝔹
v ::= λx:T.t | true | false | <integers>
```

(keeping things ASCII-easy: We occasionally write B instead of 𝔹, and Z instead of ℤ.)

"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
    - If there is no valid type, it is not in the language.
  - types could be implicit or explicit
    - We will add some explicit annotations to our language.
- type ascription operator, **:** see **STLC.hs**
  - true : $\mathbb{B}$
  - 5 : $\mathbb{Z}$
  - (Pair true 5) : $(\!(\mathbb{B},\mathbb{Z})\!)$

Sample Rule: Ty-app

$$\frac{\Gamma \vdash t_1 : T_d \rightarrow T_r \qquad \Gamma \vdash t_2 : T_d}{\Gamma \vdash (t_1\ t_2) : T_r}$$

Notes on $\lambda_{\rightarrow}$

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 Ω- and y-combinators.)
- but we can regain recursion by extending the language as before
  - fix extension will look the same.

Maintaining an environment

typechecking

- navigates subterms to determine types
- but, no substitution occurs (we're not evaluating yet)
- we will need to look at a variable and 'remember'
  at what type it was introduced, to understand each later usage

handling variables

- store their types when introduced:
  lambda parameter is in scope during the lambda body.
- all enclosing lambdas' variables/types must be tracked
- save them in a set, Γ ("gamma"), called the environment.

Maintaining an environment

How can we store this information?

(in our Haskell implementation)

- We can keep a set (or list) of pairs, **[(String, Ty)]**
  - look up variables' types
- this "environment" of all variables currently in scope is called **Gamma** (**Γ**).

Basic typing rules

- Ty-var $$\frac{(x,T)\in\Gamma}{\Gamma \vdash x : T}$$

we remember a variable's type and can look it up.

if the set Γ has (x,T) in it, then Γ can look up variable x and find its type is T

- Ty-λ $$\frac{(\Gamma,(x,T_d)) \vdash t_r : T_r}{\Gamma\vdash\ (\lambda\ x{:}T_d\ .\ t_r)\ :\ T_d \rightarrow T_r}$$

Lambdas are functions from the argument's explicitly given type to the body's found type.

If we extend Γ with (x,T_d) and that finds t_r's type to be T_r , then the original Γ derives the type of $(\lambda x{:}T_d.T_r)$ to be $T_d \rightarrow T_r$

- Ty-app $$\frac{\Gamma\vdash t_1{:}T_d\rightarrow T_r \qquad \Gamma\vdash t_2{:}T_d}{\Gamma \vdash (t_1\ t_2) : T_r}$$

if the argument's type is compatible, an application results in the func's output type.

If Γ finds $t_1{:}T_d\rightarrow T_r$, and Γ finds $t_2{:}T_d$, then Γ derives the overall type of $(t_1\ t_2)$ to be T_r

Typing rules: Booleans

- Ty-true

$$\frac{}{\vdash \text{true} : \mathbb{B}}$$

without consulting any environment, we know the type of true is $\mathbb{B}$.

- Ty-false

$$\frac{}{\vdash \text{false} : \mathbb{B}}$$

same for false.

- Ty-if

$$\frac{\Gamma \vdash a : \mathbb{B} \quad \Gamma \vdash b : T_1 \quad \Gamma \vdash c : T_1}{\Gamma \vdash \text{if } a\ b\ c : T_1}$$

If's need boolean guards, and branches of matching types.

If Γ derives a's type as $\mathbb{B}$, and Γ derives that both b and c have some type T, then Γ derives the overall type of (if a b c) as T.

Typing rules: Numbers

- Ty-$\mathbb{Z}$

$$\frac{}{\vdash \text{<\#>} : \mathbb{Z}}$$

Whole numbers are ints.

With no Γ needed, we know any literal integer is of type $\mathbb{Z}$.

- Ty-add

$$\frac{\Gamma \vdash t_1 : \mathbb{Z} \qquad \Gamma \vdash t_2 : \mathbb{Z}}{\Gamma \vdash (t_1 + t_2) : \mathbb{Z}}$$

Adding ints gives us an int.

If Γ derives that t_1 and t_2 are both of type $\mathbb{Z}$, then Γ derives (t_1+t_2) to be of type $\mathbb{Z}$.

- Ty-GT

$$\frac{\Gamma \vdash t_1 : \mathbb{Z} \;, \quad \Gamma \vdash t_2 : \mathbb{Z}}{\Gamma \vdash t_1 > t_2 \;:\; \mathbb{B}}$$

Comparing ints results in a bool.

If Γ derives t1 and t2 are both of type $\mathbb{Z}$, then G derives that t1>t2 is of type $\mathbb{B}$.

Ty-sub, Ty-mul, TyLT: similar to Ty-add and Ty-GT

see **`STLC.hs`**

Using Typing Rules – Typing Proof Trees

- We don't simplify (this isn't evaluation), 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. *This is like a roadmap of the stack while running typechecking!*

- 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 – surprisingly simple!**

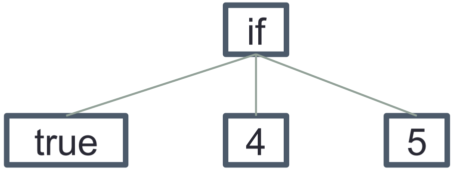
show that: (1 + 3) : $\mathbb{Z}$

$$\dfrac{\dfrac{}{\{\}\vdash 1 : \mathbb{Z}}\,\textit{Ty-}\mathbb{Z} \qquad \dfrac{}{\{\}\vdash 3 : \mathbb{Z}}\,\textit{Ty-}\mathbb{Z}}{\{\}\vdash (1 + 3) : \mathbb{Z}}\,\textit{Ty-add}$$

Using Typing Rules – Typing Proof Trees

show that: `(if true 4 5)` : $\mathbb{Z}$

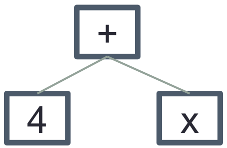
$$\dfrac{\dfrac{}{\{\}\vdash \texttt{true} : \mathbb{B}}\,\textit{Ty-true} \qquad \dfrac{}{\{\}\vdash 4 : \mathbb{Z}}\,\textit{Ty-}\mathbb{Z} \qquad \dfrac{}{\{\}\vdash 5 : \mathbb{Z}}\,\textit{Ty-}\mathbb{Z}}{\{\} \vdash \texttt{(if true 4 5)} : \mathbb{Z}}\,\textit{Ty-If}$$



Using Typing Rules – Typing Proof Trees

Begin with the knowledge that $x : \mathbb{Z}$. so, $\Gamma=\{(x,\mathbb{Z})\}$

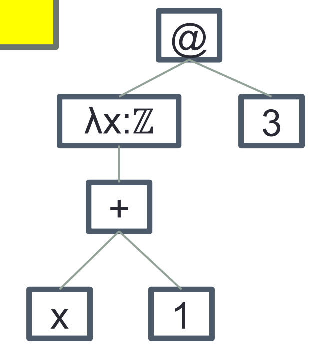
Show that: $(4 + x) : \mathbb{Z}$



$$\dfrac{\dfrac{}{\{(x,\mathbb{Z})\} \vdash 4 : \mathbb{Z}}\ \textit{Ty-}\mathbb{Z} \qquad \dfrac{}{\{(x,\mathbb{Z})\} \vdash x : \mathbb{Z}}\ \textit{Ty-Var}}{\{(x,\mathbb{Z})\} \vdash (4 + x) : \mathbb{Z}}\ \textit{Ty-Add}$$

Using Typing Rules – Typing Proof Trees

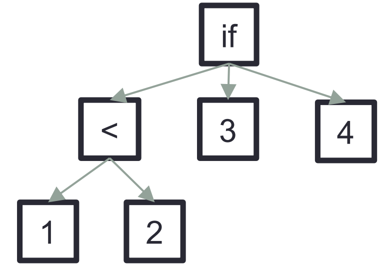
show that: $((\lambda x{:}\mathbb{Z}.x+1)\ \ 3)\ :\ \mathbb{Z}$



$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{(x,\mathbb{Z})\}\vdash x:\ \mathbb{Z}}\,\textit{Ty-Var} \qquad \dfrac{}{\{(x,\mathbb{Z})\}\vdash 1:\mathbb{Z}}\,\textit{Ty-}\mathbb{Z}}{\{(x,\mathbb{Z})\}\vdash\ x+1\ :\ \mathbb{Z}}\,\textit{Ty-Add}}{\{\}\vdash\ (\lambda x:\mathbb{Z}.x+1)\ :\ \mathbb{Z}\to\mathbb{Z}}\,\textit{Ty-}\lambda \qquad \dfrac{}{\{\}\vdash 3:\mathbb{Z}}\,\textit{Ty-}\mathbb{Z}}{\{\}\ \vdash\ ((\lambda x:\mathbb{Z}.x+1)\ 3)\ :\ \mathbb{Z}}\,\textit{Ty-App}$$

Typing Proof Tree Examples

show that: if (1<2) 3 4 : $\mathbb{Z}$

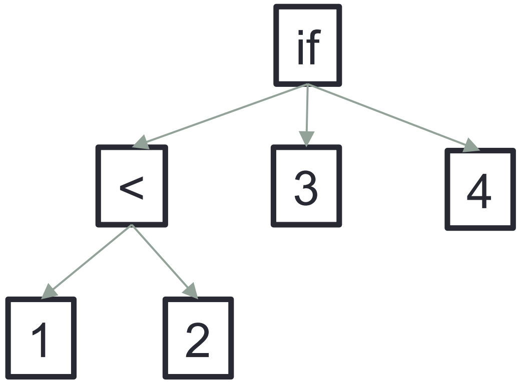


$$\dfrac{\dfrac{\dfrac{}{\{\}\vdash 1:\ \mathbb{Z}}\,\textit{Ty-}\mathbb{Z} \qquad \dfrac{}{\{\}\vdash 2:\ \mathbb{Z}}\,\textit{Ty-}\mathbb{Z}}{\{\}\vdash\ (1<2)\ :\ \mathbb{B}}\,\textit{Ty-LT} \qquad \dfrac{}{\{\}\vdash\ 3:\ \mathbb{Z}}\,\textit{Ty-}\mathbb{Z} \qquad \dfrac{}{\{\}\vdash\ 4:\ \mathbb{Z}}\,\textit{Ty-}\mathbb{Z}}{\{\}\vdash\ \texttt{if}\ (1<2)\ 3\ 4\ \ :\ \mathbb{Z}}\,\textit{Ty-If}$$

Typing Proof Tree Shape

show that: if (1<2) 3 4 : $\mathbb{Z}$

$$\dfrac{\dfrac{\dfrac{}{\{\}\vdash 1 : \mathbb{Z}}\textit{Ty-}\mathbb{Z} \quad \dfrac{}{\{\}\vdash 2 : \mathbb{Z}}\textit{Ty-}\mathbb{Z}}{\{\}\vdash (1<2) : \mathbb{B}}\textit{Ty-LT} \quad \dfrac{}{\{\}\vdash 3 : \mathbb{Z}}\textit{Ty-}\mathbb{Z} \quad \dfrac{}{\{\}\vdash 4 : \mathbb{Z}}\textit{Ty-}\mathbb{Z}}{\{\}\vdash \texttt{if (1<2) 3 4} : \mathbb{Z}}\textit{Ty-If}$$



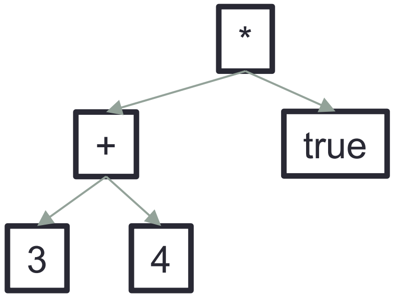
The shape of the syntax tree exactly gives you the (inverted) shape of the proof tree.

Typing Proof Tree Examples

try/**fail** to show that: **(3+4) * true : $\mathbb{Z}$**

As soon as you can show a specific requirement of a specific rule cannot be satisfied, you can stop. This might be sooner or later depending on which parts of the proof tree you work on.

$$\dfrac{\dfrac{\dfrac{}{\{\}\vdash 3:\mathbb{Z}}\,\textit{Ty-}\mathbb{Z} \qquad \dfrac{}{\{\}\vdash 4:\mathbb{Z}}\,\textit{Ty-}\mathbb{Z}}{\{\}\vdash (3+4) : \mathbb{Z}}\,\textit{Ty-Add} \qquad \dfrac{}{\{\}\vdash \text{true}:\mathbb{B}}\,\textit{Ty-true}}{\{\}\vdash (3+4) * \text{true} : \ ???}\,\textit{Ty-Mul}$$

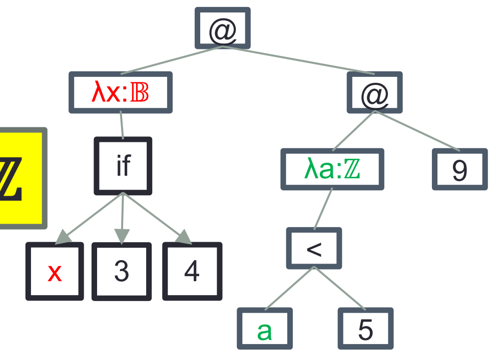


does not typecheck:

- **Ty-mul requires t_2:$\mathbb{Z}$, but we have true:$\mathbb{B}$ …**
- (Ty-mul's t_1 term does typecheck, by the way)

Name the rule that fails, and explain why.

Typing Proof Tree Examples



show that: $((\lambda x:\mathbb{B}.\ \text{if } x\ 3\ 4)\ ((\lambda a:\mathbb{Z}.\ a<5)\ 9))\ :\ \mathbb{Z}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{(x,\mathbb{B})\}\vdash x:\mathbb{B}}\,\textit{Ty-Var}\quad \dfrac{}{\{(x,\mathbb{B})\}\vdash 3:\mathbb{Z}}\,\textit{Ty-}\mathbb{Z}\quad \dfrac{}{\{(x,\mathbb{B})\}\vdash 4:\mathbb{Z}}\,\textit{Ty-}\mathbb{Z}}{\{(x,\mathbb{B})\}\vdash(\text{if } x\ 3\ 4):\mathbb{Z}}\,\textit{Ty-If}}{\{\}\vdash(\lambda x:\mathbb{B}.\text{if } x\ 3\ 4):\mathbb{B}\to\mathbb{Z}}\,\textit{Ty-}\lambda\quad \dfrac{\dfrac{\dfrac{\dfrac{}{\{(a,\mathbb{Z})\}\vdash a:\mathbb{Z}}\,\textit{Ty-Var}\quad \dfrac{}{\{(a,\mathbb{Z})\}\vdash 5:\mathbb{Z}}\,\textit{Ty-}\mathbb{Z}}{\{(a,\mathbb{Z})\}\vdash(a<5):\mathbb{B}}\,\textit{Ty-LT}}{\{\}\vdash(\lambda a:\mathbb{Z}.a<5):\mathbb{Z}\to\mathbb{B}}\,\textit{Ty-}\lambda\quad \dfrac{}{\{\}\vdash 9:\mathbb{Z}}\,\textit{Ty-}\mathbb{Z}}{\{\}\vdash((\lambda a:\mathbb{Z}.a<5)\ 9)\ :\ \mathbb{B}}\,\textit{Ty-App}}{\{\}\vdash\ ((\lambda x:\mathbb{B}.\text{if } x\ 3\ 4)\ ((\lambda a:\mathbb{Z}.a<5)\ 9))\ :\ \mathbb{Z}}\,\textit{Ty-App}$$

Practice Problems – Typing Proof Trees

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

1. $((\lambda x{:}\mathbb{Z}.x)\ ((\lambda n{:}\mathbb{Z}.n)\ 10))$
2. $((\lambda x{:}\mathbb{Z}.\text{if true } x\ (x+1))\ 5)$
3. $(\lambda f{:}\mathbb{Z}\rightarrow\mathbb{Z}.\ \lambda x{:}\mathbb{Z}.f\ x)$
4. $(((\lambda x{:}\mathbb{Z}.\ \lambda y{:}\mathbb{Z}.x+y)\ 5)\ 8)$

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

5. $(3\ 5)$
6. $((\lambda x{:}\mathbb{Z}.x+1)\ \text{true})$

Practice Problems - encodings

Now that we have our language, let's use it!

- Other than adding types to lambdas, this is just as easy as in the untyped lambda calculus.
  - We just label the type of our lambdas' arguments along the way

Encode these:

- `and  ::` $\mathbb{B} \longrightarrow \mathbb{B} \longrightarrow \mathbb{B}$
- `or   ::` $\mathbb{B} \longrightarrow \mathbb{B} \longrightarrow \mathbb{B}$
- `ge   ::` $\mathbb{Z} \longrightarrow \mathbb{Z} \longrightarrow \mathbb{B}$ *(greater or equal than)*
- `inc  ::` $\mathbb{Z} \longrightarrow \mathbb{Z}$
- `max3 ::` $\mathbb{Z} \longrightarrow \mathbb{Z} \longrightarrow \mathbb{Z} \longrightarrow \mathbb{Z}$

Extending $\lambda_{\rightarrow}$

- Terms/values extended as before
  → often need explicit types, e.g. λx:**T**.t
- Often add one new type to T ::= for each extension
- Each term gets exactly one typing rule
  → no substitution, so **easier** than evaluation rules!
  → must maintain Γ, so **trickier** than evaluation rules!
  → usually invoke typechecking (:) on all subterms,
  used to claim overall term's type

Practice: Pairs Extension

Language additions:

t ::= … | pair t t | fst t | snd t

v ::= … | pair t t

T ::= … | ⟪ T , T ⟫

Evaluation Rules:

(same as before)

Typing Rules:

(one per new term)

Ty-Pair: $$\frac{\Gamma \vdash t_1 : T_1, \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \text{pair } t_1\ t_2 \ : ⟪ T_1, T_2 ⟫}$$

Ty-Fst: $$\frac{\Gamma \vdash t : ⟪ T_1, T_2 ⟫}{\Gamma \vdash \text{fst } t \ : T_1}$$

Ty-Snd: $$\frac{\Gamma \vdash t : ⟪ T_1, T_2 ⟫}{\Gamma \vdash \text{snd } t \ : T_2}$$

Practice Problems – Pairs + Typing Proof Trees

Draw typing proof trees to find these expressions' types:

1. pair true 4
2. fst (pair 3 true)
3. snd p, with $\Gamma=\{(p, ⦇\mathbb{B},\mathbb{Z}⦈)\}$.
4. $\lambda p:⦇\mathbb{Z},\mathbb{B}⦈.$ if (snd p) (fst p) 0
5. (pair $(x:\mathbb{Z}.x+1)$ $(\lambda f: \mathbb{Z}\longrightarrow\mathbb{Z}.f\ 1)$)

Fails typechecking:

1. fst $(\lambda x:\mathbb{Z}.$pair x x)

Practice: Recursion Extension

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

Language addition: t ::= … | fix t *(note: fix t is not a value!)*

Evaluation Rule:
(same as before, with type ascription)

E-Fix:
$$\frac{}{\text{fix } (\lambda\ x{:}\mathbf{T}\ .\ t) \longrightarrow t\ [x \mapsto (\text{fix } (\lambda\ x{:}\mathbf{T}\ .\ t))]}$$

Typing Rule:
(one per new term)

Ty-Fix:
$$\frac{\Gamma \vdash t : T \longrightarrow T}{\Gamma \vdash \text{fix } t : T}$$

Practice Problems - Recursion

Draw proof trees to find these expressions' types:

1. fix (λself:$\mathbb{Z}\longrightarrow\mathbb{Z}$. λn:$\mathbb{Z}$. if (n<2) 1 (n*(self (n-1)))
   - Just sketch the bottom two levels to see **Ty-Fix** in use.

Encode these no-lists-involved things:

2. factorial (shown above)
3. fibonacci

Practice: Lists Extension (p.1/2)

Language additions:

t ::= … | nil **T** | cons t t | head t | tail t | isnil t

T ::= … | ⟦ T ⟧

v ::= … | nil T | cons t t

note: nil needs a type! Why?

Evaluation Rules:

E-head: $$\frac{t \rightarrow t'}{\text{head } t \rightarrow \text{head } t'}$$

E-head-cons: $$\frac{}{\text{head}(\text{cons } t_1\ t_2) \rightarrow t_1}$$

E-tail: $$\frac{t \rightarrow t'}{\text{tail } t \rightarrow \text{tail } t'}$$

E-tail-cons: $$\frac{}{\text{tail}(\text{cons } t_1\ t_2) \rightarrow t_2}$$

E-isnil: $$\frac{t \rightarrow t'}{\text{isnil } t \rightarrow \text{isnil } t'}$$

E-isnil-true: $$\frac{}{\text{isnil }(\text{nil } T) \rightarrow \text{true}}$$

E-isnil-false: $$\frac{}{\text{isnil }(\text{cons } t_1\ t_2) \rightarrow \text{false}}$$

Practice: Lists Extension (p.2/2)

Evaluation Rules:
(same as before)

Language additions:

t ::= … | nil **T** | cons t t | head t | tail t | isnil t
T ::=... | ⟦ T ⟧
v ::= ... | nil T | cons t t

nil T: the T should be the overall list-type, e.g. nil⟦ℤ⟧

Typing Rules: *(one per new term)*

Ty-nil: $$\frac{}{\vdash \text{nil } [\![T]\!] : [\![T]\!]}$$

Ty-cons: $$\frac{\Gamma \vdash t_1 : T, \quad \Gamma \vdash t_2 : [\![T]\!]}{\Gamma \vdash \text{cons } t_1\ t_2 : [\![T]\!]}$$

Ty-head: $$\frac{\Gamma \vdash t : [\![T]\!]}{\Gamma \vdash \text{head } t : T}$$

Ty-tail: $$\frac{\Gamma \vdash t : [\![T]\!]}{\Gamma \vdash \text{tail } t : [\![T]\!]}$$

Ty-isnil: $$\frac{\Gamma \vdash t : [\![T]\!]}{\Gamma \vdash \text{isnil } t : \mathbb{B}}$$

Practice Problems - Lists

Draw proof trees to find these expressions' types:

1. head (if true (cons 5 (nil $[\![\mathbb{Z}]\!]$)) (nil $[\![\mathbb{Z}]\!]$)))
2. head (tail (cons 10 (cons 12 (cons 13 (nil $[\![\mathbb{Z}]\!]$)))))

Encode these:

3. length:: $[\![\mathbb{Z}]\!] \longrightarrow \mathbb{Z}$
4. map:: $(\mathbb{Z}\longrightarrow\mathbb{Z}) \longrightarrow [\![\mathbb{Z}]\!] \longrightarrow [\![\mathbb{Z}]\!]$
5. nth :: $[\![\mathbb{Z}]\!] \longrightarrow \mathbb{Z} \longrightarrow \mathbb{Z}$ *could fail at evaluation!*

Thoughts

- With no extensions, (with only `t::=x|λx:T.t|(t t)`) $\lambda_{\rightarrow}$ is degenerate (it has no values). Why?
- evaluation should *preserve types* – a term's value should not change types due to further evaluation.
  $\longrightarrow$ true for $\lambda_{\rightarrow}$ as presented
- **erasure**: after typechecking, we can erase all types in $\lambda_{\rightarrow}$ and evaluation is unaffected. That's neat!
  - 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*

Logic	concept	Type Theory	concept
proposition	• a statement (may be T/F)	types	group of values
$P \supset Q$	• given proof P, make proof of Q	type $P \to Q$	function from P to Q
$P \wedge Q$	• stmt that P and Q are true	type $P \times Q$	product type (e.g. tuple)
$P \vee Q$	• stmt that P or Q is true	type $P + Q$	union type (e.g. Either a b)
proof of P	• way to show truth of P	term of type P	way to construct value
P is provable	• claim: P is true	type P inhabited	claim: elt of P exists