



# **Type Systems Part 1: Simply-Typed $\lambda$ -Calculus**

**CIS352 — Spring 2024**

**Kris Micinski**

# Type Systems

A **type system** assigns each source fragment with a given **type**: a specification of how it will behave

Type systems include **rules**, or **judgements** that tells us how we compositionally build types for larger fragments from smaller fragments

The **goal** of a type system is to **rule out** programs that would exhibit run time type errors!

Which of the following should be allowed to run?

`(λ (g) (g 5))`

`((λ (f x) (f x)) g)`

`(λ (f) (+ (f 1) (if (= 0 (string-length (f 1))) 1 0)))`

# Which of the following should be allowed to run?

`(λ (g) (g 5))`

Nothing **obviously** wrong, in absence of more knowledge about `g`

`((λ (f x) (f x)) g)`

You **cannot** call this lambda, it will **necessarily** result in an error

`(λ (f) (+ (f 1) (if (= 0 (string-length (f 1))) 1 0)))`

Can't work if `f` is pure (not stateful): `(f 1)` can't return a number and a string

Type systems will give us a formal (in the sense of having a form we can write down) description to when a program has a type.

A type is a rough approximation of the value's behavior. For example, the type `int` might represent the type of all integers, while the type `int -> int` would be the functions from values of type `int` to values of type `int`

(Preview of where we're going)

We'll be able to use a type system to be able to deduce that, because x and y are passed to +, they **must** be ints (+ constraints its arguments to be ints)

```
;; OCaml, not Racket  
((fun x y -> (x + y)) 1 2);;
```

Further, OCaml can **infer** that  $x$  and  $y$  **must** be ints using a process called **type inference**, which is common in a variety of forms today (Rust, TypeScript, Haskell, ...)

```
;; OCaml, not Racket  
# (fun x y -> (x + y));;  
- : int -> int -> int = <fun>
```

Further, OCaml can **infer** that  $x$  and  $y$  **must** be ints using a process called **type inference**, which is common in a variety of forms today (Rust, TypeScript, Haskell, ...)

```
;; OCaml, not Racket  
# (fun x y -> (x + y));;  
- : int -> int -> int = <fun>
```



OCaml's type system is an extension of the Simply-Typed Lambda Calculus to recursive types (algebraic data types). Its type inference is able to catch quite nuanced issues

**Question:** is this code okay?

```
# (fun x f -> (if (f 3) then ((f x) + 5) else 8));;
```

OCaml's type system is an extension of the Simply-Typed Lambda Calculus to recursive types (algebraic data types). Its type inference is able to catch quite nuanced issues

```
# (fun x f -> (if (f 3) then ((f x) + 5) else 8));;
```

**Error:** This expression has type `bool` but an  
expression was expected of type  
`int`

In type theory, a subexpression has a **type** when there exists some **proof** according to a formally-defined typing derivation.

You will learn how to write proofs for typing derivations in the Simply-Typed Lambda Calculus, a small core of a type system for a functional language.

# Simply-Typed $\lambda$ -calculus

STLC is a restriction of the untyped  $\lambda$ -calculus  
(It is a restriction in the sense that not all terms are well-typed.)

Expressions in STLC, assuming  $t$  is a type (we'll show this soon):

$$\begin{array}{l} e ::= (\text{lambda } (x : t) \ e) \\ \quad | (e \ e) \\ \quad | (\text{prim } e \ e) \\ \quad | x \\ \quad | n \\ \quad | (e : t) \end{array}$$

All lambdas **must** be annotated with their type

Optionally, any subexpression may be **annotated** with a type

$$\text{prim} ::= + \mid * \mid \dots$$

```

;; Expressions are ifarith, with several special builtins
(define (expr? e)
  (match e
    ;; Variables
    [(? symbol? x) #t]
    ;; Literals
    [(? bool-lit? b) #t]
    [(? int-lit? i) #t]
    ;; Applications
    [`(, (? expr? e0) , (? expr? e1)) #t]
    ;; Annotated expressions
    [`(, (? expr? e) : , (? type? t)) #t]
    ;; Annotated lambdas
    [`(lambda (, (? symbol? x) : , (? type? t)) , (? expr? e)) #t]))

```

The ***simply typed*** lambda calculus is a type system built on top of a small kernel of the lambda calculus

Crucially, STLC is *less expressive* than the lambda calculus (e.g., we cannot type  $\Omega$ ,  $Y$ , or  $U$ !)

In practice, STLC's restrictions make it unsuitable for serious programming—but it is the basis for many modern type systems in real languages (e.g., OCaml, Rust, Swift, Haskell, ...)

Terms *inhabit* types  
(via the typing judgement)

## Term Syntax

$$e ::= (\text{lambda } (x : t) \ e) \\ \quad | \ (e \ e) \\ \quad | \ (\text{prim } e \ e) \\ \quad | \ x \\ \quad | \ n \\ \quad | \ (e : t)$$
$$\text{prim} ::= + \mid * \mid \dots$$

## Type Syntax

$$t ::= \text{int} \\ \quad | \ \text{bool} \\ \quad | \ t \rightarrow t$$

## Term Syntax

```
e ::= (lambda (x : t) e)
      | (e e)
      | (prim e e)
      | x
      | n
      | (e : t)
```

```
prim ::= + | * | ...
```

## Type Syntax

```
t ::= int
      | bool
      | t -> t
```

Function Types





## Term Syntax

```
e ::= (lambda (x : t) e)
      | (e e)
      | (prim e e)
      | x
      | n
      | (e : t)
```

```
prim ::= + | * | ...
```

## Type Syntax

```
t ::= int
      | bool
      | t -> t
```

### Examples...

```
bool -> int
int -> (int -> int)
(int -> int) -> int
(bool -> (int -> bool)) -> int
```

- Type checking happens hierarchically
- Literals (0, #f) have their obvious types
- More complex forms (lambda, apply) require us to type subexpressions

For example, let's say we have this lambda, which we want to type check:

$$\left( \lambda(x : \text{num}) \text{ (if (x = 0) x (+ x 1))} \right)$$

First we see the input type is int. Assuming x is int, we type check the body (an if). We see both sides of the if result in a number, so we know the lambda's output is also a number.

Thus, the type is `int -> int`

Notice that in STLC, all lambdas **must** bind their argument by naming a type explicitly. Thus, the following is **not** an STLC term.

However, the term has an infinite number of possible types:

$$\left( \lambda(x) \text{ (if \#f (x 5) (x 8))} \right)$$

The term may be **monomorphized** by instantiating once for each type  $T$  such that  $T$  is something like...

$$\left( \lambda(x : T_0 \rightarrow T_1) \text{ (if \#f (x 5) (x 8))} \right)$$

**Question:** why  $T_0 \rightarrow T_1$  rather than any type  $T$ ? **Answer:**  $x$  is applied (must be function)

**Exercise:** Write three possible monomorphizations, what is the type of the lambda as a whole?

The fact that lambdas must be annotated with a type makes typing easy: parameters are the only true source of non-local control in the lambda calculus, and represent the only ambiguity in type checking

$$\left( \lambda(x : \text{num} \rightarrow \text{num}) \text{ (if \#f (x 5) (x 8))} \right)$$

One possible monomorphization



### *Bad thought experiment*

`(if #f (x 5) (x 8))`

Let's say `x` is the Racket lambda:

`(λ (x) (if (< x 6) #t 5))`

Now, when `x` is less than 6, we return something of type `bool`; but otherwise, we return something of type `int`.

`(+ 3 (if #f (x 5) (x 8)))`

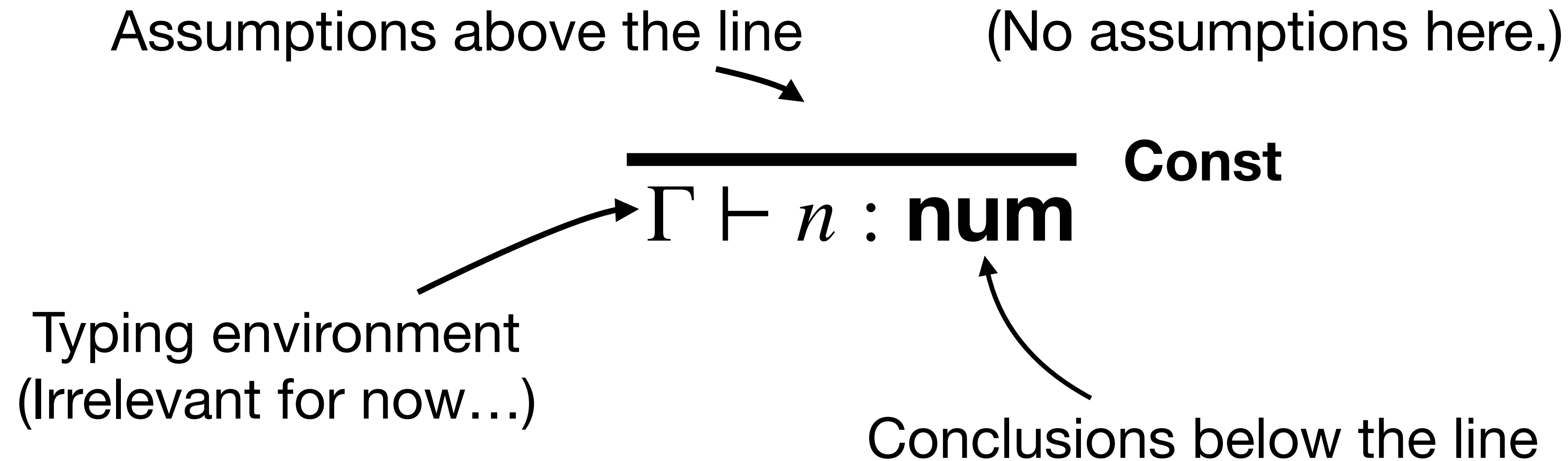
In this case, the `+` operation works as long as `(x 8)` returns a `int`, but what if `(x 8)` returns a `bool`?

*A few examples...*

$$(\lambda(x : \text{num}) (\lambda (y : \text{bool}) y)) : \text{num} \rightarrow \text{bool} \rightarrow \text{bool}$$
$$\left( \lambda(x : \text{num} \rightarrow \text{num}) (x \ 5) \right) : (\text{num} \rightarrow \text{num}) \rightarrow \text{num}$$
$$\left( \lambda(x : \text{num} \rightarrow \text{num}) (\text{if } \#f \ (x \ 5) \ (x \ 8)) \right) : (\text{num} \rightarrow \text{num}) \rightarrow \text{num}$$

# STLC Typing Rules

Type rules are written in ***natural-deduction*** style



The rule reads “in any typing environment  $\Gamma$ , we may conclude the literal number  $n$  has type `int`”

$$\frac{}{\Gamma \vdash n : \mathbf{num}} \quad \mathbf{Const}$$



# Variable Lookup

We assume a **typing environment** which maps variables to their types

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \quad \mathbf{Var}$$

If  $x$  maps to type  $t$  in  $\Gamma$ , we may conclude that  $x$  has type  $t$  under the type environment  $\Gamma$

**Exercise:** using the **Var** rule, complete the proof

---

$\{x \mapsto (\mathbf{num} \rightarrow \mathbf{num}), y \mapsto \mathbf{bool}\} \vdash x : ???$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \quad \mathbf{Var}$$

# ***Solution***

$$\frac{\{x \mapsto (\mathbf{num} \rightarrow \mathbf{num}), y \mapsto \mathbf{bool}\}(x) = \mathbf{num} \rightarrow \mathbf{num}}{\{x \mapsto (\mathbf{num} \rightarrow \mathbf{num}), y \mapsto \mathbf{bool}\} \vdash x : (\mathbf{num} \rightarrow \mathbf{num})} \mathbf{Var}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \mathbf{Var}$$

# Typing Functions

If, assuming  $x$  has type  $t$ , you can conclude the body  $e$  has type  $t'$ , then the whole lambda has type  $t \rightarrow t'$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda (x : t) e) : t \rightarrow t'} \quad \text{Lam}$$

If, assuming  $x$  has type  $t$ , you can conclude the body  $e$  has type  $t'$ , then the whole lambda has type  $t \rightarrow t'$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda (x : t) e) : t \rightarrow t'} \quad \text{Lam}$$

Notice: if we didn't have type  $t$  here, we would have to **guess**, which could be quite hard. We will have to do this when we move to allow *type inference*

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda (x : t) e) : t \rightarrow t'} \quad \text{Lam}$$

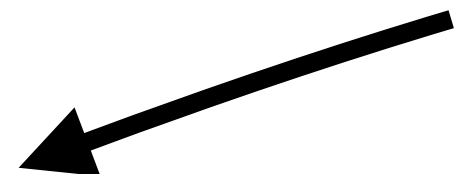
**Example:** let's use the Lam rule to ascertain the type of the following expression.

---

(lambda (x : int) 1)

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda (x : t) e) : t \rightarrow t'} \quad \text{Lam}$$

Start with the empty environment (since this term is closed)



$$\Gamma = \{\} \vdash (\text{lambda } (x : \text{int}) \ 1) : ? \rightarrow ?$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda (x : t) e) : t \rightarrow t'} \quad \text{Lam}$$

$$\Gamma = \{\} \vdash \overline{(\text{lambda } (x : \text{int}) 1) : t \rightarrow t'}$$

We **suppose** there are two types,  $t$  and  $t'$ , which will make the derivation work.



Because  $x$  is tagged, it must be **int**

$$\frac{\{x \mapsto \mathbf{num}\} \vdash 1 : t'}{\Gamma = \{\} \vdash (\text{lambda } (x : \text{int}) \ 1) : \mathbf{num} \rightarrow t'}$$

We **suppose** there are two types,  $t$  and  $t'$ , which will make the derivation work.

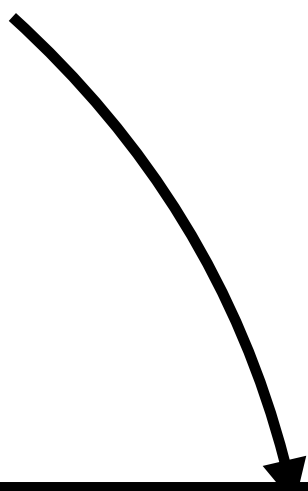
The **Const** rule allows us to conclude  $1 : \mathbf{int}$

$$\frac{\frac{}{\{x \mapsto \mathbf{num}\} \vdash 1 : t'}}{\Gamma = \{\} \vdash (\text{lambda } (x : \mathbf{int}) \ 1) : \mathbf{num} \rightarrow t'} \text{Lam}$$

We **suppose** there are two types,  $t$  and  $t'$ , which will make the derivation work.

So  $t' = \text{int}$

Notice: **Const** demands no subgoals


$$\frac{\frac{}{\{x \mapsto \mathbf{num}\} \vdash 1 : \mathbf{num}} \text{Const}}{\Gamma = \{\} \vdash (\text{lambda } (x : \text{int}) \ 1) : \mathbf{num} \rightarrow \mathbf{num}} \text{Lam}$$


# Function Application

$$\frac{\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e \ e') : t'} \quad \text{App}$$

# Function Application

If (under Gamma),  $e$  has type  $t \rightarrow t'$

And  $e'$  (its argument) has type  $t$


$$\frac{\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e \ e') : t'} \quad \text{App}$$

Then the application of  $e$  to  $e'$  results in a  $t'$

# Our type system so far...

$$\begin{array}{c} \text{Const} \\ \hline \Gamma \vdash n : \mathbf{num} \end{array} \qquad \begin{array}{c} \text{True} \quad (\text{Also False}) \\ \hline \Gamma \vdash \#t : \mathbf{bool} \end{array} \qquad \begin{array}{c} \text{Var} \\ \hline \Gamma(x) = t \\ \Gamma \vdash x : t \end{array}$$
$$\begin{array}{c} \Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t \\ \hline \Gamma \vdash (e \ e') : t' \end{array} \quad \text{App}$$
$$\begin{array}{c} \Gamma[x \mapsto t] \vdash e : t' \\ \hline \Gamma \vdash (\lambda (x : t) \ e) : t \rightarrow t' \end{array} \quad \text{Lam}$$

Almost everything! What about builtins?

- Almost everything! What about builtins?
- A few ways to handle this:
  - Add ***pairs*** to our language
  - Builtins accept pairs

$$\Gamma_l = \{ + : (\mathbf{num} \times \mathbf{num}) \rightarrow \mathbf{num}, \dots \}$$

- Or, we could assume that primitives are simply curried—in that case we would have, e.g.,  $((+ 1) 2)$  and then...

$$\Gamma_l = \{ + : \mathbf{num} \rightarrow (\mathbf{num} \rightarrow \mathbf{num}), \dots \}$$

- ***Our exercise does this!!***

Two possibilities (pairs/currying)

$e ::= (\text{lambda } (x : t) e)$   
 $\quad \quad \quad | (e e)$   
 $\quad \quad \quad | (\text{prim } (e, e)) ; \text{pairs}$   
 $\quad \quad \quad | ((\text{prim } e) e) ; \text{curry}$   
 $\quad \quad \quad | x$   
 $\quad \quad \quad | n$   
 $\quad \quad \quad | (e : t)$

$\text{prim} ::= + \mid * \mid \dots$

# Practice Derivations

Write derivations of the following expressions...



$((\lambda (x : \text{int}) x) 1)$

$$\frac{}{\Gamma \vdash n : \mathbf{num}} \quad \mathbf{Const} \qquad \frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t} \quad \mathbf{Var}$$

$$\frac{\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e e') : t'} \quad \mathbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda (x : t) e) : t \rightarrow t'} \quad \mathbf{Lam}$$

$((\lambda (x : \text{int}) x) 1)$

<b>Var</b>	<hr/>	
	$\{x \mapsto \mathbf{num}\} \vdash x : \mathbf{num}$	
<b>Lam</b>	<hr/>	
	$\{\} \vdash (\lambda (x : \mathbf{num}) x) : \mathbf{num} \rightarrow \mathbf{num}$	
<b>App</b>	<hr/>	
	$\{\} \vdash ((\lambda (x : \mathbf{num}) x) 1) : \mathbf{num}$	
		<hr/>
		<b>Const</b>
		$\{\} \vdash 1 : \mathbf{num}$

$((\lambda (f : \text{int} \rightarrow \text{int}) (f\ 1)) (\lambda (x : \text{int}) x))$

$$\frac{}{\Gamma \vdash n : \mathbf{num}} \quad \mathbf{Const} \qquad \frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t} \quad \mathbf{Var}$$

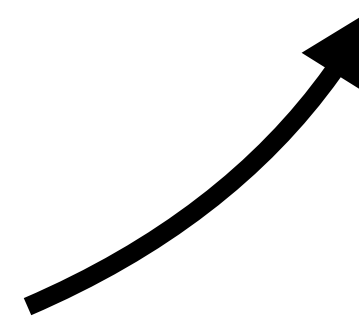
$$\frac{\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'} \quad \mathbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda (x : t) e) : t \rightarrow t'} \quad \mathbf{Lam}$$

# Typability in STLC

Not all terms can be given types...

$(\lambda (f : \text{int} \rightarrow \text{int}) (f f))$



It is impossible to write a derivation for the above term!

$f$  is  $\text{int} \rightarrow \text{int}$  but would **need** to be  $\text{int}$ !

# Typability

Not all terms can be given types...

$$\begin{array}{c} ((\lambda (f) (f f)) \\ (\lambda (f) (f f))) \end{array}$$

It is **impossible** to write a derivation for  $\Omega$ !

Consider what would happen if  $f$  were:

- $\text{int} \rightarrow \text{int}$
- $(\text{int} \rightarrow \text{int}) \rightarrow \text{int}$

**Always just out of reach...**

# Type Checking

Type checking asks: given this fully-typed term, is the type checking done correctly?

$$((\lambda (x:\text{int}) x:\text{int}) : \text{int} \rightarrow \text{int})$$

In practice, as long as we annotate arguments (of  $\lambda$ s) with specific types, we can elide the types of variables, literals, and applications

The “simply typed” nature of STLC means that type inference is very simple...

## Exercise

For each of the following expressions, do they type check?  
I.e., is it possible to construct a typing derivation for them?  
If so, what is the type of the expression?

$(\lambda (f : \text{int} \rightarrow \text{int} \rightarrow \text{int}) ((f\ 2)\ 3)\ 4))$

$((\lambda (f : \text{int} \rightarrow \text{int}) f) (\lambda (x:\text{int}) (\lambda (x:\text{int}) x)))$

## *Solution*

*Neither* type checks.

This subexpression results in **int**,  
which cannot be applied.

$(\lambda (f : \text{int} \rightarrow \text{int} \rightarrow \text{int}) ((f\ 2)\ 3)\ 4))$

$((\lambda (f : \text{int} \rightarrow \text{int}) f) (\lambda (x:\text{int}) (\lambda (x:\text{int}) x)))$



## *Solution*

*Neither* type checks.

```
(λ (f : int -> int -> int) (((f 2) 3) 4))
```

```
((λ (f : int -> int) f) (λ (x:int) (λ (x:int) x)))
```

This binder *demand*s its argument is of type `int -> int`,  
but its argument is *really* of type `int -> int -> int`

In the case of fully-annotated STLC, we never have to *guess* a type

In STLC, type *inference* is no harder than type *checking*

Our type checker will be **syntax-directed**

Next lecture, we will look at type inference for **un-annotated** STLC

- This will require generating, and then solving, constraints

The basic approach is to observe that each of the rules applies to a different *form*

For example, if we hit *any* application expression ( $e\ e'$ ), we know that we *have* to use the **App** rule

Thus, we write our type checker as a structurally-recursive function over the input expression.

$$\frac{}{\Gamma \vdash n : \mathbf{num}} \mathbf{Const} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \mathbf{Var}$$

$$\frac{\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'} \mathbf{App}$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda (x : t) e) : t \rightarrow t'} \mathbf{Lam}$$

```
;; Synthesize a type for e in the environment env
;; Returns a type
(define (synthesize-type env e)
  (match e
    ;; Literals
    [(? integer? i) 'int]
    [(? boolean? b) 'bool]
```

**Const**

$$\Gamma \vdash n : \mathbf{num}$$

Recognizing literals is easy

```

;; Synthesize a type for e in the environment env
;; Returns a type
(define (synthesize-type env e)
  (match e
    ;; Literals
    [(? integer? i) 'int]
    [(? boolean? b) 'bool]
    ;; Look up a type variable in an environment
    [(? symbol? x) (hash-ref env x)]

```

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \quad \textbf{Var}$$

```

;; Synthesize a type for e in the environment env
;; Returns a type
(define (synthesize-type env e)
  (match e
    ;; Literals
    [(? integer? i) 'int]
    [(? boolean? b) 'bool]
    ;; Look up a type variable in an environment
    [(? symbol? x) (hash-ref env x)]
    ;; Lambda w/ annotation
    [`(lambda (,x : ,A) ,e)
     `(,A -> ,(synthesize-type (hash-set env x A) e))])

```

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda (x : t) e) : t \rightarrow t'} \quad \text{Lam}$$

```

;; Synthesize a type for e in the environment env
;; Returns a type
(define (synthesize-type env e)
  (match e
    ;; Literals
    [(? integer? i) 'int]
    [(? boolean? b) 'bool]
    ;; Look up a type variable in an environment
    [(? symbol? x) (hash-ref env x)]
    ;; Lambda w/ annotation
    [`(lambda (,x : ,A) ,e)
     `(,A -> ,(synthesize-type (hash-set env x A) e)))]
    ;; Arbitrary expression
    [`(,e : ,t) (let ([e-t (synthesize-type env e)])
                  (if (equal? e-t t)
                      t
                      (error (format "types ~a and ~a are different" e-t t)))))]
  )

```

We haven't written this rule yet—but  
 notice how the  $t$ 's are implicitly unified  
 (i.e., asserted to be the same) in the rule

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (e : t) : t} \text{ Chk}$$

;; Synthesize a type for e in the environment env

;; Returns a type

(define (synthesize-type env e)

(match e

;; Literals

[(? integer? i) 'int]

[(? boolean? b) 'bool]

;; Look up a type variable in an environment

[(? symbol? x) (hash-ref env x)]

;; Lambda w/ annotation

[`(lambda (,x : ,A) ,e)

  `(:,A -> ,(synthesize-type (hash-set env x A) e))]

;; Arbitrary expression

[`(:,e : ,t) (let ([e-t (synthesize-type env e)])

          (if (equal? e-t t)

              t

              (error (format "types ~a and ~a are different" e-t t)))))]

;; Application

[`(:,e1 ,e2)

  (match (synthesize-type env e1)

    [`(:,A -> ,B)

      (let ([t-2 (synthesize-type env e2)])

        (if (equal? t-2 A)

            B

            (error (format "types ~a and ~a are different" A t-2)))))])))]

$$\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t$$

$$\Gamma \vdash (e \ e') : t'$$

**App**