Type Systems Part 1: Simply-Typed λ-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 of an expression's runtime form

A type is a rough approximation of the expression'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);;
```

This process called **type inference**, and is common in modern languages (Examples include Rust, TypeScript, Haskell, OCaml …)

You can think of type inference as an *always correct CoPilot*, but the correctness also means expressivity is limited only to properties about which the type system is designed to reason.

A type inference system means that you write as many annotations as you want—the compiler figures out what you mean and tells you when it hits an inconsistency.

Question: is this code okay?

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

A type inference system means that you write as many annotations as you want—the compiler figures out what you mean and tells you when it hits an inconsistency.

```
# (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 λ-calculus

STLC is a restriction of the untyped λ-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):

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

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

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

```
prim ::= + | * | …
```

```
;; 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]
    ;; Anotated 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 Ω, 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 ::= (lambda (x : t) e)
    | (e e)
    | (prim e e)
    | x
    | n
    | (e : t)

prim ::= + | * | …
```

Type Syntax

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

$$\Big(\lambda(x : \text{int})\ \big(\text{if } (x = 0)\ x\ (+\ x\ 1)\big)\Big)$$

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`

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

$$\left(\lambda(x : \mathsf{int} \to \mathsf{int})\ \left(\mathsf{if}\ \#\mathsf{f}\ (x\ 5)\ (x\ 8)\right)\right)$$

We know both the input and output type

Bad thought experiment

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

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`.

$$\left(+\ \ 3\ \left(\text{if \#f } (x\ 5)\ (x\ 8)\right)\right)$$

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

We could write a **union type**, but STLC will make things simpler: the true/false branch **must** have the same type.

A few examples…

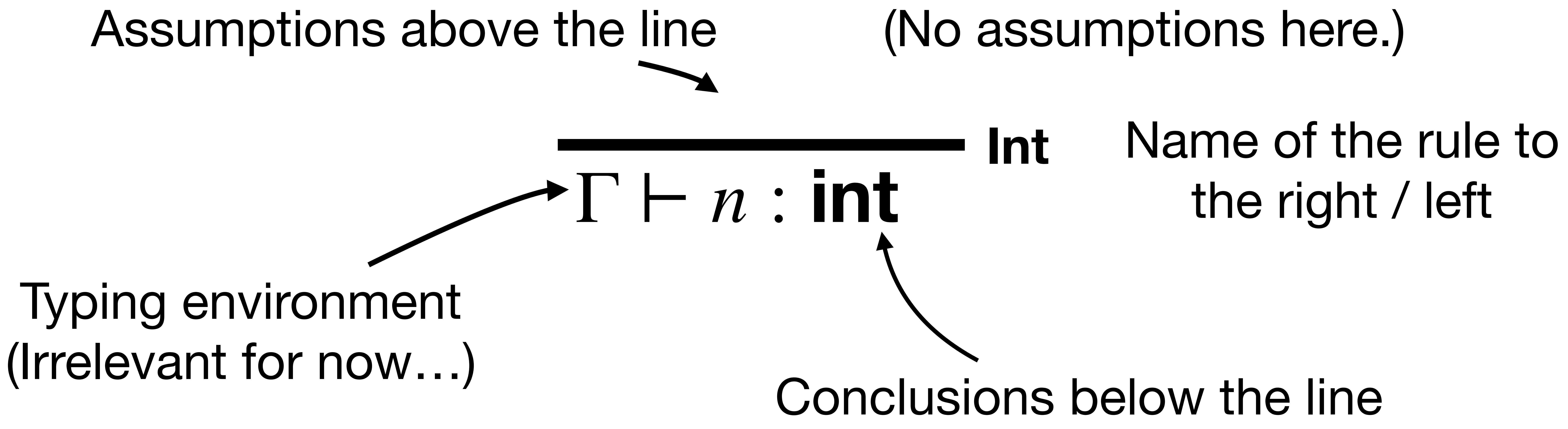
$$\big(\lambda(x : \mathsf{int})\ (\lambda\ (y : \mathsf{bool})\ y)\big)\ : \mathsf{int} \to \mathsf{bool} \to \mathsf{bool}$$

$$\Big(\lambda(x : \mathsf{int} \to \mathsf{int})\ (x\ 5))\ : (\mathsf{int} \to \mathsf{int}) \to \mathsf{num}$$

$$\Big(\lambda(x : \mathsf{int} \to \mathsf{int})\ \big(\mathsf{if}\ \#\mathsf{f}\ (x\ 5)\ (x\ 8)\big)\Big)\ : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$$

STLC Typing Rules

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



Assumptions above the line

(No assumptions here.)

$$\frac{}{\Gamma \vdash n : \textbf{int}}\ \textbf{Int}$$

Name of the rule to the right / left

Typing environment
(Irrelevant for now…)

Conclusions below the line

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

$$\frac{}{\Gamma \vdash n : \textbf{int}}\ \textbf{Int}$$

Variable Lookup

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

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

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

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

$$\frac{\quad}{\{x \mapsto (\textbf{int} \rightarrow \textbf{int}), y \mapsto \textbf{bool}\} \vdash x : ???}$$

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

Solution

$$\dfrac{\{x \mapsto (\textbf{int} \to \textbf{int}), y \mapsto \textbf{bool}\}(x) = \textbf{int} \to \textbf{int}}{\{x \mapsto (\textbf{int} \to \textbf{int}), y \mapsto \textbf{bool}\} \vdash x : (\textbf{int} \to \textbf{int})}\ \textbf{Var}$$

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

Typing Functions

If, assuming x has type x, you can conclude the body e has type t’, then the whole lambda has type t → t’

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

If, assuming x has type x, you can conclude the body e has type t’, then the whole lambda has type t → t’

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

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*

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

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

$$\frac{}{\texttt{(lambda (x : int) 1)}}$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

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

$$\Gamma = \{\} \vdash \frac{\qquad}{\texttt{(lambda (x : int) 1)}} : ? \to ?$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

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

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

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

$$\frac{\{x \mapsto \textbf{int}\} \vdash 1 : t'}{\Gamma = \{\} \vdash \texttt{(lambda (x : int) 1)} : \textbf{int} \rightarrow t'}$$

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

The **Int** rule allows us to conclude 1 : **int**

$$\dfrac{\dfrac{}{\{x \mapsto \textbf{int}\} \vdash 1 : t'}}{\Gamma = \{\} \vdash \texttt{(lambda (x : int) 1)} : \textbf{int} \to t'}\ \textbf{Lam}$$

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

So t’ = **int**

Notice: **Int** demands no subgoals

$$\dfrac{\dfrac{}{\{x \mapsto \textbf{int}\} \vdash 1 : \textbf{int}}\ \textbf{Int}}{\Gamma = \{\} \vdash \texttt{(lambda (x : int) 1)} : \textbf{int} \to \textbf{int}}\ \textbf{Lam}$$

Function Application

$$\frac{\Gamma \vdash e : t \to t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

Function Application

If (under Gamma), e has type `t -> t’`

And `e’` (its argument) has type t

$$\frac{\Gamma \vdash e : t \to t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ \ e') : t'}\ \textbf{App}$$

Then the application of e to `e’` results in a `t’`

Our type system so far…

$$\frac{}{\Gamma \vdash n : \textbf{int}}\ \textbf{Int}$$

(Also **False**)

$$\frac{}{\Gamma \vdash \textbf{\#t} : \textbf{bool}}\ \textbf{True}$$

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

$$\frac{\Gamma \vdash e : t \rightarrow t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

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

Almost everything! What about builtins?

- Almost everything! What about builtins?
- A few ways to handle this:
  - Add ***pairs*** to our language
    - We'll see this next time

$$\Gamma_\iota = \{ + : (\textbf{num} \times \textbf{num}) \rightarrow \textbf{num}, \ldots \}$$

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

$$\Gamma_\iota = \{ + : \textbf{num} \rightarrow (\textbf{num} \rightarrow \textbf{num}), \ldots \}$$

Two possibilities (pairs/currying)

```
e ::= (lambda (x : t) e)
    | (e e)
    | (prim (e, e)) ; pairs
    | ((prim e) e)  ; curry
    | x
    | n
    | (e : t)

prim ::= + | * | …
```

Practice Derivations

Write derivations of the following expressions…

```
((λ (x : int) x) 1)
```

$$\frac{}{\Gamma \vdash n : \textbf{int}} \ \textbf{Int} \qquad \frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t} \ \textbf{Var}$$

$$\frac{\Gamma \vdash e : t \to t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'} \ \textbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda\, (x : t)\ e) : t \to t'} \ \textbf{Lam}$$

```
((λ (x : int) x) 1)
```

$$\dfrac{\dfrac{\dfrac{}{\{x \mapsto \textbf{int}\} \vdash x : \textbf{int}}\;\textbf{Var}}{\{\,\} \vdash (\lambda\,(x : \textbf{int})\ x) : \textbf{int} \to \textbf{int}}\;\textbf{Lam} \qquad \dfrac{}{\{\,\} \vdash 1 : \textbf{int}}\;\textbf{Int}}{\{\,\} \vdash \Big((\lambda\,(x : \textbf{int})\ x)\ 1\Big) : \textbf{int}}\;\textbf{App}$$

```
((λ (f : int -> int) (f 1)) (λ (x : int) x))
```

$$\frac{}{\Gamma \vdash n : \textbf{int}}\ \textbf{Int} \qquad \frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t}\ \textbf{Var}$$

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

Typability in STLC

Not all terms can be given types…

```
(λ (f : int -> int) (f f))
```

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

f is `int->int` but would **need** to be `int`!

Typability

Not all terms can be given types…

```
((λ (f) (f f))
 (λ (f) (f f)))
```

It is **impossible** to write a derivation for Ω!

Consider what would happen if `f` were:

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

Always just out of reach…

Type **Checking**

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

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

In practice, as long as we annotate arguments (of λs) 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?

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

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

Solution

Neither *type checks.*

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

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

((λ (f : int -> int) f) (λ (x:int) (λ (x: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 ***demands*** 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*

$$\frac{}{\Gamma \vdash n : \textbf{int}}\ \textbf{Int} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ \textbf{Var}$$

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

$$\frac{\Gamma \vdash e : t \rightarrow t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

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

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

```
;; 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]
```

$$\frac{}{\Gamma \vdash n : \textbf{int}}\ \textbf{Int}$$

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)]
```

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

```
;; 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))]
```

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

```
;; 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

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

```
;; 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))))])]))
```

$$\frac{\Gamma \vdash e : t \to t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$