

Type Declarations

$$\Gamma \vdash \langle \text{num} \rangle : \text{num} \quad [\dots \langle \text{id} \rangle \leftarrow \tau \dots] \vdash \langle \text{id} \rangle : \tau$$
$$\Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash \text{false} : \text{bool}$$
$$\Gamma \vdash \mathbf{e}_1 : \text{num} \quad \Gamma \vdash \mathbf{e}_2 : \text{num}$$

$$\Gamma \vdash \{+ \mathbf{e}_1 \ \mathbf{e}_2\} : \text{num}$$
$$\Gamma \vdash \mathbf{e}_1 : \text{bool} \quad \Gamma \vdash \mathbf{e}_2 : \tau_0 \quad \Gamma \vdash \mathbf{e}_3 : \tau_0$$

$$\Gamma \vdash \{\text{if } \mathbf{e}_1 \ \mathbf{e}_2 \ \mathbf{e}_3\} : \tau_0$$
$$\Gamma[\langle \text{id} \rangle \leftarrow \tau_1] \vdash \mathbf{e} : \tau_0$$

$$\Gamma \vdash \{\text{fun } \{ \langle \text{id} \rangle : \tau_1 \} \ \mathbf{e}\} : (\tau_1 \rightarrow \tau_0)$$
$$\Gamma \vdash \mathbf{e}_0 : (\tau_1 \rightarrow \tau_0) \quad \Gamma \vdash \mathbf{e}_1 : \tau_1$$

$$\Gamma \vdash \{\mathbf{e}_0 \ \mathbf{e}_1\} : \tau_0$$

Type Inference

$$\Gamma \vdash \langle \text{num} \rangle : \text{num} \quad [\dots \langle \text{id} \rangle \leftarrow \tau \dots] \vdash \langle \text{id} \rangle : \tau$$
$$\Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash \text{false} : \text{bool}$$
$$\Gamma \vdash \mathbf{e}_1 : \text{num} \quad \Gamma \vdash \mathbf{e}_2 : \text{num}$$

$$\Gamma \vdash \{ + \mathbf{e}_1 \ \mathbf{e}_2 \} : \text{num}$$
$$\Gamma \vdash \mathbf{e}_1 : \text{bool} \quad \Gamma \vdash \mathbf{e}_2 : \tau_0 \quad \Gamma \vdash \mathbf{e}_3 : \tau_0$$

$$\Gamma \vdash \{ \text{if } \mathbf{e}_1 \ \mathbf{e}_2 \ \mathbf{e}_3 \} : \tau_0$$
$$\Gamma [\langle \text{id} \rangle \leftarrow \tau_1] \vdash \mathbf{e} : \tau_0$$

$$\Gamma \vdash \{ \text{fun } \{ \langle \text{id} \rangle \} \ \mathbf{e} \} : (\tau_1 \rightarrow \tau_0)$$
$$\Gamma \vdash \mathbf{e}_0 : (\tau_1 \rightarrow \tau_0) \quad \Gamma \vdash \mathbf{e}_1 : \tau_1$$

$$\Gamma \vdash \{ \mathbf{e}_0 \ \mathbf{e}_1 \} : \tau_0$$

Do the rules still work? (Yes.)

$$\frac{[x \leftarrow num] \ x : num \qquad [x \leftarrow num] \ 1 : num}{[x \leftarrow num] \ \{+ \ x \ 1\} : num}$$

$$[x \leftarrow num] \ \{+ \ x \ 1\} : num$$

$$\emptyset \vdash \{\text{fun } \{x\} \ \{+ \ x \ 1\}\} : (num \rightarrow num)$$

But how do we implement them now?

Type Inference

- **Type inference** is the process of inserting type annotations where the programmer omits them
- We'll use explicit question marks, to make it clear where types are omitted

```
{ fun {x : ?} {+ x 1} }
```

Type Inference

- **Type inference** is the process of inserting type annotations where the programmer omits them
- We'll use explicit question marks, to make it clear where types are omitted

```
{ fun {x : ?} {+ x 1} }
```

```
<typeExpr> ::= num  
            | bool  
            | (<typeExpr> -> <typeExpr>)  
            | ?
```

Type Inference

```
{ fun {x : ?} {+ x 1} }
```

Type Inference

```
{ fun {x : ?} {+ x 1} }
```

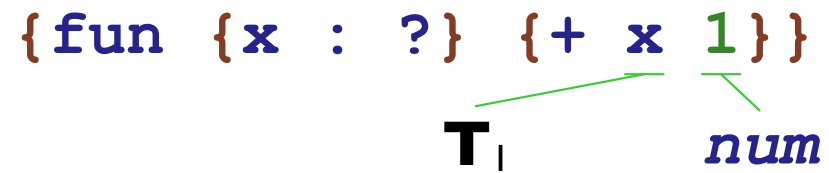
T₁



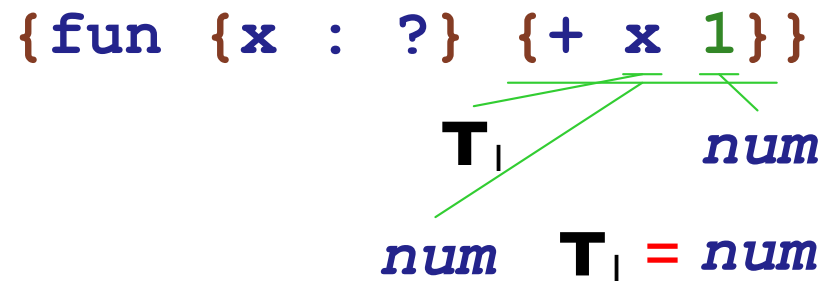
Type Inference

`{ fun {x : ?} {+ x 1} }`

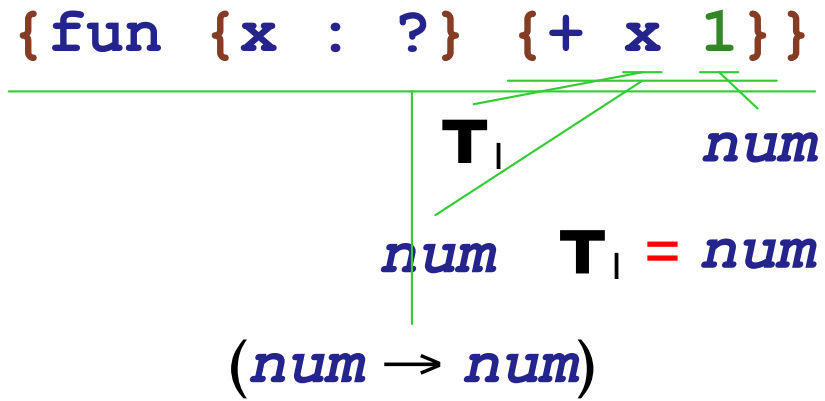
T₁ *num*



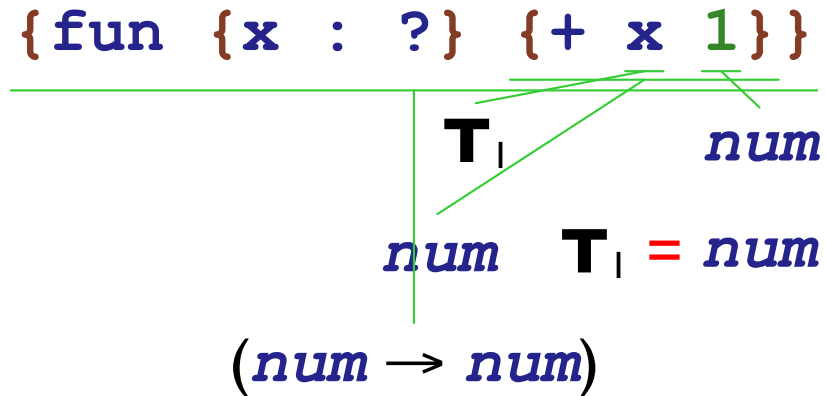
Type Inference



Type Inference

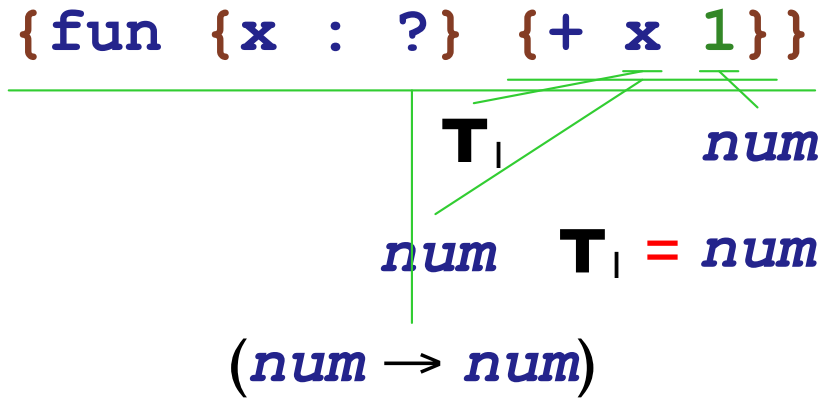


Type Inference



- Create a new type variable for each ?
- Change type comparison to install type equivalences

Type Inference



`{fun {x : ?} {if true 1 x}}`

Type Inference

`{ fun {x : ?} {+ x 1} }`

\mathbf{T}_1 *num*

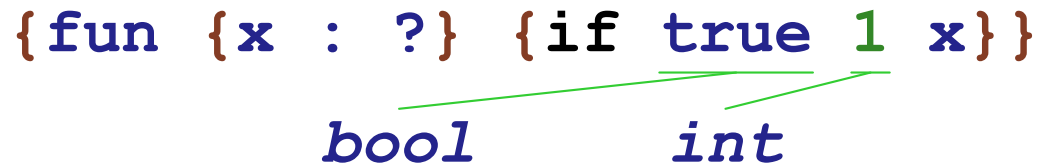
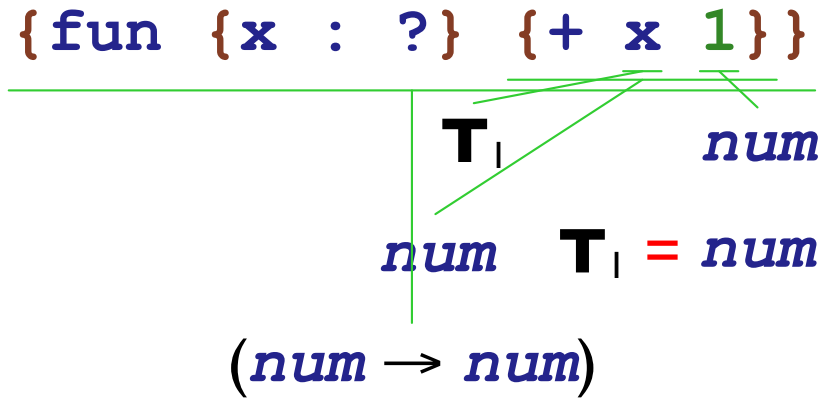
num $\mathbf{T}_1 = \mathit{num}$

$(\mathit{num} \rightarrow \mathit{num})$

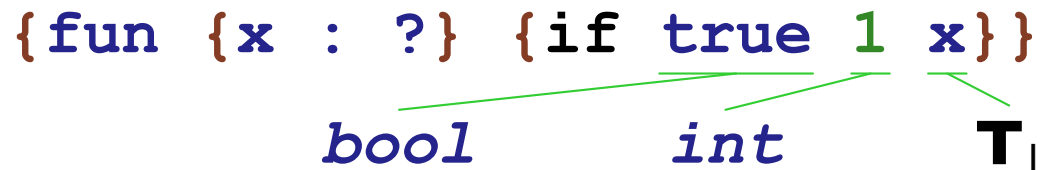
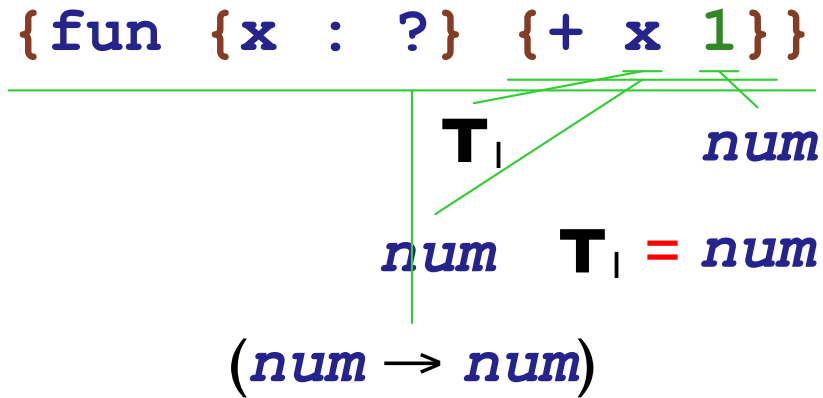
`{ fun {x : ?} {if true 1 x} }`

bool

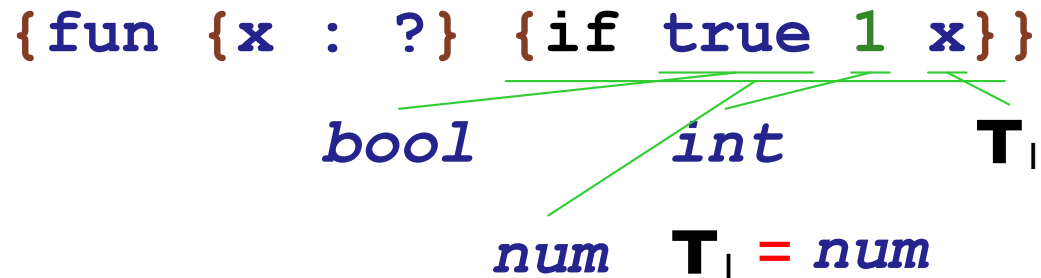
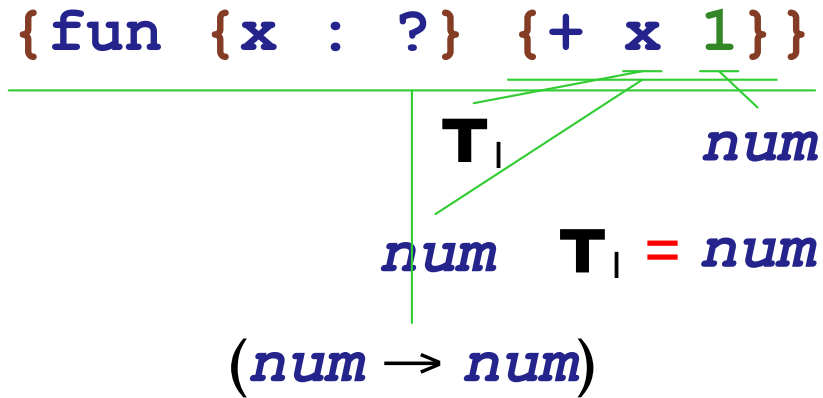
Type Inference



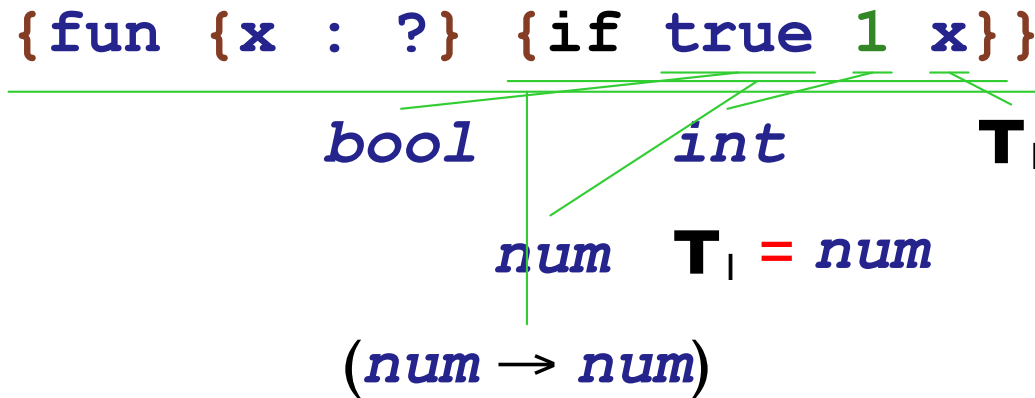
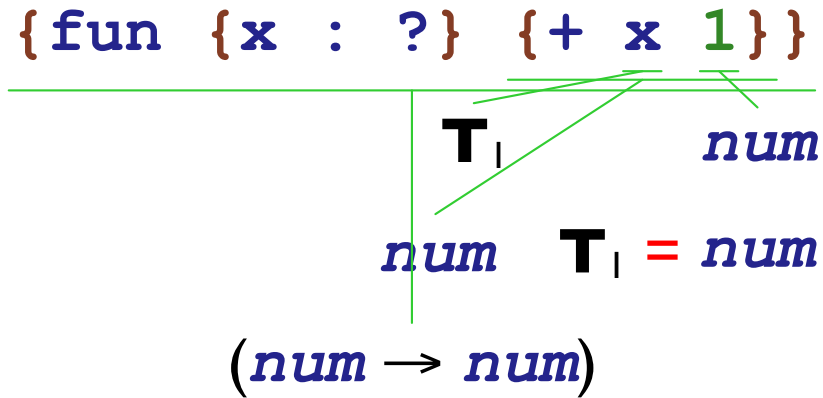
Type Inference



Type Inference



Type Inference



Type Inference: Impossible Cases

```
{fun {x : ?} {if x 1 x}}
```

Type Inference: Impossible Cases

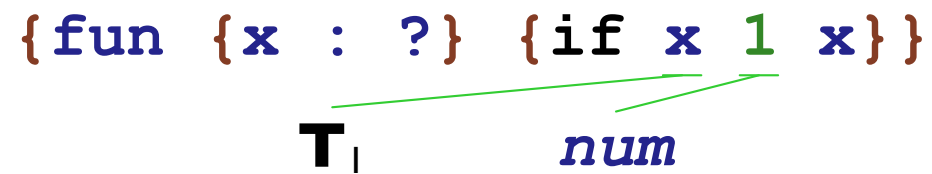
```
{fun {x : ?} {if x 1 x}}
```

T₁

Type Inference: Impossible Cases

```
{fun {x : ?} {if x 1 x}}
```

T₁ *num*



Type Inference: Impossible Cases

`{fun {x : ?} {if x 1 x}}`

T_1 *num* T_1

Type Inference: Impossible Cases

```
{fun {x : ?} {if x 1 x}}
```

T_1 *num* T_1

no type: T_1 can't be both *bool* and *num*

Type Inference: Many Cases

```
{ fun {y : ?} y }
```

Type Inference: Many Cases

```
{ fun {y : ?} y }
```

T₁

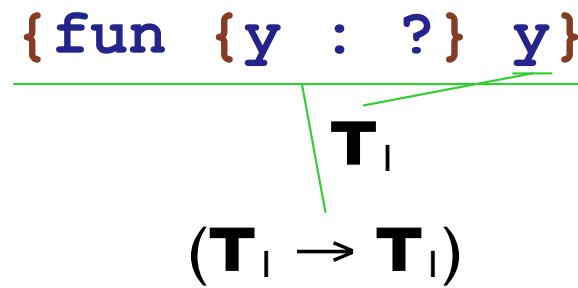


Type Inference: Many Cases

`{ fun {y : ?} y }`

T₁

(T₁ → T₁)



Type Inference: Many Cases

$$\frac{\{\text{fun } \{y : ?\} y\}}{\mathbf{T}_1}$$
$$(\mathbf{T}_1 \rightarrow \mathbf{T}_1)$$

- Sometimes, more than one type works
 - (*num* → *num*)
 - (*bool* → *bool*)
 - ((*num* → *bool*) → (*num* → *bool*))

so the type checker leaves variables in the reported type

Type Inference: Function Calls

```
{{fun {y : ?} y} {fun {x : ?} {+ x 1}}}}
```

Type Inference: Function Calls

{fun {y : ?} y} {fun {x : ?} {+ x 1}}

(**T**₁ → **T**₁)

Type Inference: Function Calls

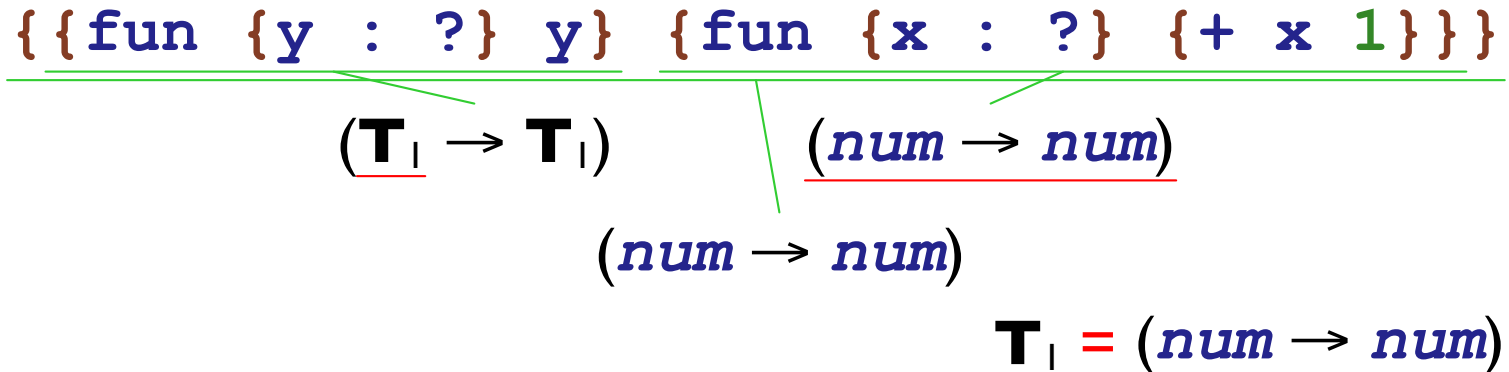
$\{\{\text{fun } \{y : ?\} y\} \text{ fun } \{x : ?\} \{+ x 1\}\}\}$
 $(\mathbf{T}_1 \rightarrow \mathbf{T}_1)$ $(\text{num} \rightarrow \text{num})$

Type Inference: Function Calls

$\{\{\text{fun } \{y : ?\} y\} \text{ fun } \{x : ?\} \{+ x 1\}\}\}$
 $(\mathbf{T}_1 \rightarrow \mathbf{T}_1)$ $(\underline{\text{num}} \rightarrow \underline{\text{num}})$

$\mathbf{T}_1 = (\text{num} \rightarrow \text{num})$

Type Inference: Function Calls



Type Inference: Function Calls

```
{fun {y : ?} {y 7}}
```


Type Inference: Function Calls

```
{fun {y : ?} {y 7}}
```

T_i

Type Inference: Function Calls

`{fun {y : ?} {y 7}}`

T_i *num*

Type Inference: Function Calls

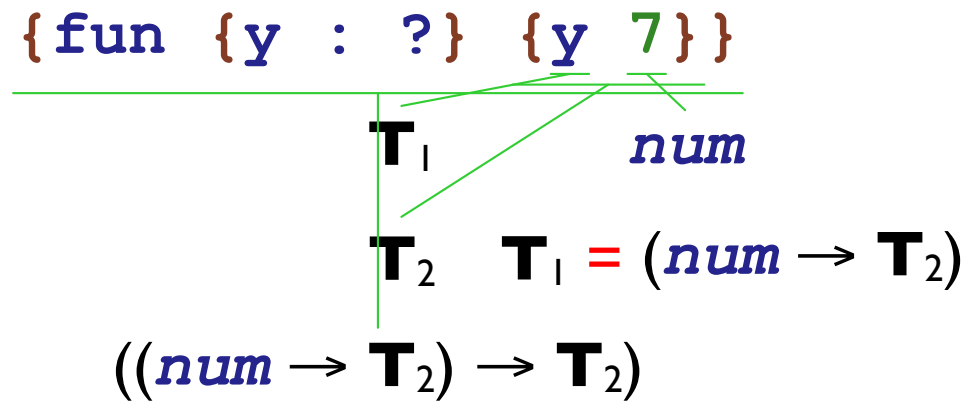
{fun {y : ?} {y 7}}

\mathbf{T}_1

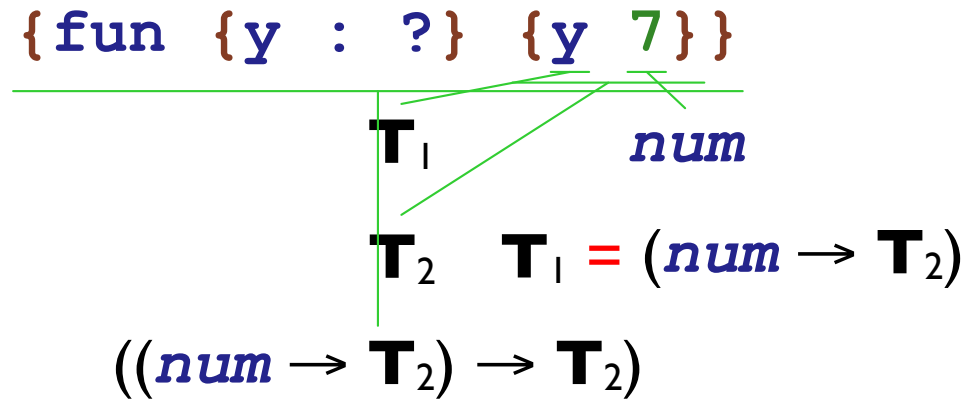
num

$\mathbf{T}_2 \quad \mathbf{T}_1 = (\mathit{num} \rightarrow \mathbf{T}_2)$

Type Inference: Function Calls



Type Inference: Function Calls



- In general, create a new type variable record for the result of a function call

Type Inference: Cyclic Equations

```
{fun {x : ?} {x x}}
```

Type Inference: Cyclic Equations

```
{fun {x : ?} {x x}}
```

T₁



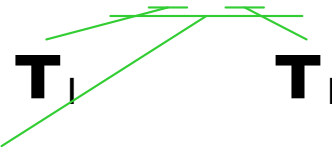
Type Inference: Cyclic Equations

`{ fun {x : ?} {x x} }`

T₁ **T₁**

Type Inference: Cyclic Equations

{fun {x : ?} {x x}}



no type: \mathbf{T}_1 can't be $(\mathbf{T}_1 \rightarrow \dots)$

- \mathbf{T}_1 can't be *int*
- \mathbf{T}_1 can't be *bool*
- Suppose \mathbf{T}_1 is $(\mathbf{T}_2 \rightarrow \mathbf{T}_3)$
 - \mathbf{T}_2 must be \mathbf{T}_1
 - So we won't get anywhere!

Type Inference: Cyclic Equations

{fun {x : ?} {x x}}

\mathbf{T}_1 \mathbf{T}_1

no type: \mathbf{T}_1 can't be $(\mathbf{T}_1 \rightarrow \dots)$

The **occurs check**:

- When installing a type equivalence, make sure that the new type for \mathbf{T} doesn't already contain \mathbf{T}

Type Unification

Unify a type variable \mathbf{T} with a type τ_2 :

- If \mathbf{T} is set to τ_1 , unify τ_1 and τ_2
- If τ_2 is already equivalent to \mathbf{T} , succeed
- If τ_2 contains \mathbf{T} , then fail
- Otherwise, set \mathbf{T} to τ_2 and succeed

Unify a type τ_1 to type τ_2 :

- If τ_2 is a type variable \mathbf{T} , then unify \mathbf{T} and τ_1
- If τ_1 and τ_2 are both *num* or *bool*, succeed
- If τ_1 is $(\tau_3 \rightarrow \tau_4)$ and τ_2 is $(\tau_5 \rightarrow \tau_6)$, then
 - unify τ_3 with τ_5
 - unify τ_4 with τ_6
- Otherwise, fail

TIFAE Grammar

```
<TIFAE> ::= <num>
          | {+ <TIFAE> <TIFAE>}
          | {- <TIFAE> <TIFAE>}
          | <id>
          | {fun {<id> : <TE>} <TIFAE>}
          | {<TIFAE> <TIFAE>}
          | {if0 <TIFAE> <TIFAE> <TIFAE>}
          | {rec {<id> : <TE> <TIFAE>} <TIFAE>}
```

```
<TE> ::= num
       | (<TE> -> <TE>)
       | ?
```



Representing Expressions

```
(define-type TFAE
  [num (n : number)]
  [add (l : TFAE)
       (r : TFAE)]
  [sub (l : TFAE)
       (r : TFAE)]
  [id (name : symbol)]
  [fun (name : symbol)
       (t : TE) ; different
       (body : TFAE)]
  [app (rator : TFAE)
       (rand : TFAE)])
```

Representing Type Variables

```
(define-type Type
  [numT]
  [boolT]
  [arrowT (arg : Type)
          (result : Type)]
  [varT (is : (boxof MaybeType))])

(define-type TE      (define-type MaybeType
  [numTE]           [none]
  [boolTE]          [some (t : Type)])
  [arrowTE
   (arg : TE)
   (result : TE)]
  [guesSTE])
```

Parsing Types

```
(define parse-type : (TE -> Type)
  (lambda (te)
    (type-case TE te
      [numTE () (numT)]
      [boolTE () (boolT)]
      [arrowTE (d r)
               (arrowT (parse-type d)
                       (parse-type r))]
      [guessTE () (varT (box (none)))])))
```

Type Unification

```
(define unify! : (Type Type -> ())
  (lambda (t1 t2)
    (type-case Type t1
      [varT (b) (type-case MaybeType (unbox b)
                    [some (t1-2) (unify! t1-2 t2)]
                    [none ()
                     (type-case Type t2
                       [varT (b2)
                        (type-case MaybeType (unbox b2)
                          [some (t2-2) (unify! t1 t2-2)]
                          [none () (if (eq? b b2)
                                       (values)
                                       (begin
                                         (set-box!
                                          b
                                          (some t2))
                                         (values))))))]
                       [else ...])]]])
      [numT () ...]
      [boolT () ...]
      [arrowT (d1 r1) ...]))))
```


Type Unification

```
(define unify! : (Type Type -> ())
  (lambda (t1 t2)
    (type-case Type t1
      [varT (b) (type-case MaybeType (unbox b)
                    [some (t1-2) (unify! t1-2 t2)]
                    [none ()
                     (type-case Type t2
                       [varT (b2)
                        ...]
                       [else (if (occurs? b t2)
                                  (error 'type-check "failed")
                                  (begin
                                     (set-box! b (some t2))
                                     (values))))))]
      [numT () ...]
      [boolT () ...]
      [arrowT (d1 r1) ...])))
```

Type Unification

```
(define unify! : (Type Type -> ())
  (lambda (t1 t2)
    (type-case Type t1
      [varT (b) ...]
      [numT () (type-case Type t2
                  [varT (b) (unify! t2 t1)]
                  [numT () (values)]
                  [else (error 'type-check "failed")])]
      [boolT () (type-case Type t2
                  [varT (b) (unify! t2 t1)]
                  [boolT () (values)]
                  [else (error 'type-check "failed")])]
      [arrowT (d1 r1) (type-case Type t2
                       [varT (b) (unify! t2 t1)]
                       [arrowT (d2 r2)
                                (begin (unify! d1 d2)
                                       (unify! r1 r2))]
                       [else (error 'type-check "failed")]])))))
```

Occurs Check

```
(define occurs? : ((boxof MaybeType) Type -> boolean)
  (lambda (b t)
    (type-case Type t
      [varT (b2) (or (eq? b b2)
                     (type-case MaybeType (unbox b2)
                       [none () false]
                       [some (t2-2) (occurs? b t2-2)]))]
      [numT () false]
      [arrowT (d r) (or (occurs? b d)
                        (occurs? b r))]))))
```

TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [num (n) (numT)]
      ...)))
```

TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [add (l r)
           ... (typecheck l env) ...
           ... (typecheck r env) ...]
      ...)))
```

TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [add (l r) (begin
                  (unify! (typecheck l env) (numT) l)
                  (unify! (typecheck r env) (numT) r)
                  (numT))])
      ...)))
```

TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [id (name) (get-type name env)]
      [fun (name te body)
        (local [(define arg-type (parse-type te))]
          (arrowT arg-type
                  (typecheck body (aBind name
                                          arg-type
                                          env))))))
      ...))
```

TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [app (fn arg)
           ... (typecheck fn env) ...
           ... (typecheck arg env) ...]
      ...)))
```


TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [app (fn arg)
        (local [(define result-type (varT (box (none))))])
          ... (arrowT (typecheck arg env)
                      result-type)
          ... (typecheck fn env) ...])
      ...)))
```

TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [app (fn arg)
        (local [(define result-type (varT (box (none))))])
          (begin
            (unify! (arrowT (typecheck arg env)
                           result-type)
                    (typecheck fn env)
                    fn)
            result-type))]
      ...)))
```

TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [if0 (test-expr then-expr else-expr)
        ... (typecheck test-expr env) ...
        ... (typecheck then-expr env) ...
        ... (typecheck else-expr env) ...]
      ...)))
```

TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [if0 (test-expr then-expr else-expr)
        (begin
          (unify! (typecheck test-expr env) (numT) test-expr)
          ... (typecheck then-expr env) ...
          ... (typecheck else-expr env) ...)]
      ...)))
```

TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [if0 (test-expr then-expr else-expr)
        (begin
          (unify! (typecheck test-expr env) (numT) test-expr)
          (local [(define test-ty
                    (typecheck then-expr env))]
            (begin
              (unify! test-ty
                (typecheck else-expr env)
                else-expr)
              test-ty)))]
      ...)))
```

TIFAE Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [rec (name ty rhs-expr body-expr)
        (local [(define rhs-ty (parse-type ty))
                (define new-ds (aBind name
                                      rhs-ty
                                      env))]
          (begin
            (unify! rhs-ty (typecheck rhs-expr new-ds) rhs-expr)
            (typecheck body-expr new-ds))))])
    ...)))
```