

Type Inference

Type Declarations

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

$$\Gamma \vdash \{ + \mathbf{e}_1 \ \mathbf{e}_2 \} : \textit{number}$$
$$\Gamma \vdash \mathbf{e}_1 : \textit{boolean} \quad \Gamma \vdash \mathbf{e}_2 : \tau_0 \quad \Gamma \vdash \mathbf{e}_3 : \tau_0$$

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

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

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

Type Inference

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

$$\Gamma \vdash \{ + \mathbf{e}_1 \ \mathbf{e}_2 \} : \textit{number}$$
$$\Gamma \vdash \mathbf{e}_1 : \textit{boolean} \quad \Gamma \vdash \mathbf{e}_2 : \tau_0 \quad \Gamma \vdash \mathbf{e}_3 : \tau_0$$

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

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

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

Do the rules still work? (Yes.)

$$\frac{[x \leftarrow \textit{number}] \ x : \textit{number} \quad [x \leftarrow \textit{number}] \ 1 : \textit{number}}{[x \leftarrow \textit{number}] \ \{+ \ x \ 1\} : \textit{number}}$$

$$\frac{[x \leftarrow \textit{number}] \ \{+ \ x \ 1\} : \textit{number}}{\emptyset \vdash \{\textit{fun} \ \{x\} \ \{+ \ x \ 1\}\} : (\textit{number} \rightarrow \textit{number})}$$

But how do we implement them now?

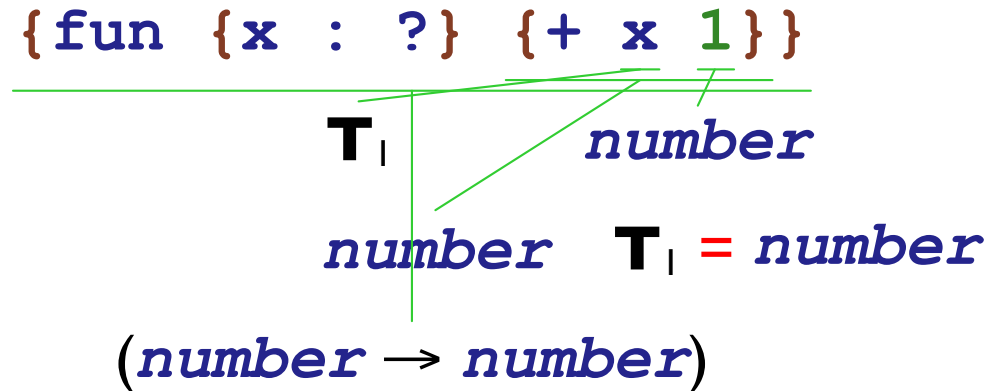
Type Inference

- **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}}
```

```
<TE> ::= number  
      | boolean  
      | (<TE> -> <TE>)  
      | ?
```

Type Inference



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

Type Inference

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

\mathbf{T}_1

number

number $\mathbf{T}_1 = \textit{number}$

(number → number)

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

boolean

number

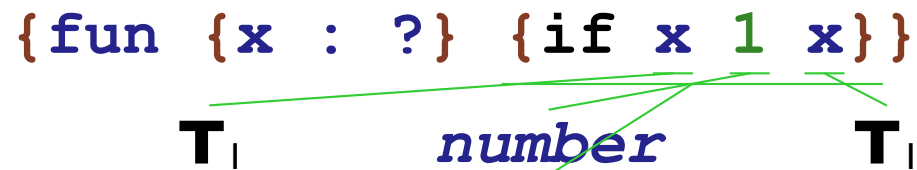
\mathbf{T}_1

number $\mathbf{T}_1 = \textit{number}$

(number → number)

Type Inference: Impossible Cases

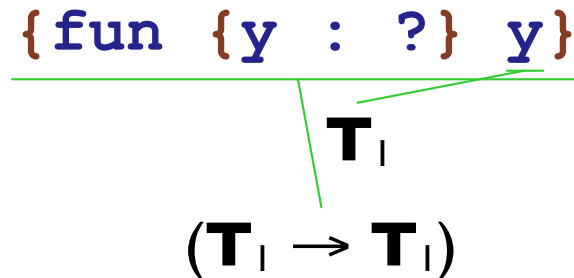
```
{fun {x : ?} {if x 1 x}}
```



T_1 *number* T_1

no type: T_1 can't be both *boolean* and *number*

Type Inference: Many Cases



- Sometimes, more than one type works
 - (*number* \rightarrow *number*)
 - (*boolean* \rightarrow *boolean*)
 - ((*number* \rightarrow *boolean*) \rightarrow (*number* \rightarrow *boolean*))

so the type checker leaves variables in the reported type

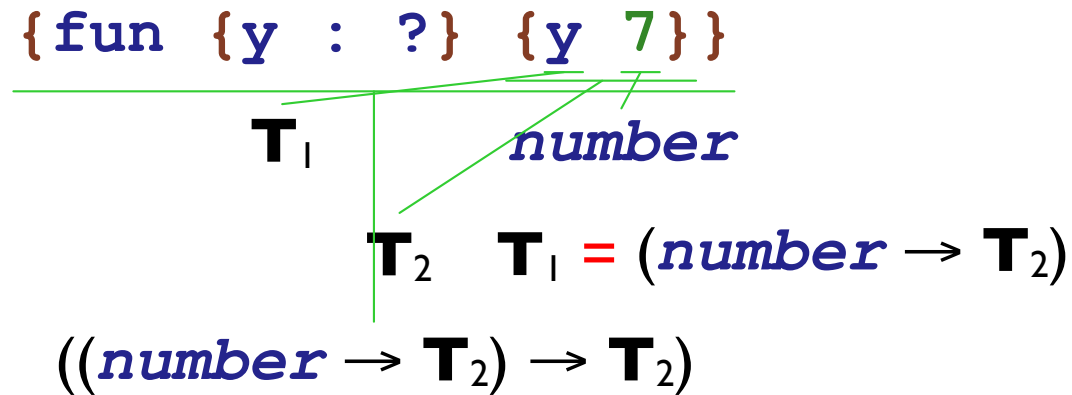
Type Inference: Function Calls

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

$(\text{number} \rightarrow \text{number})$

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

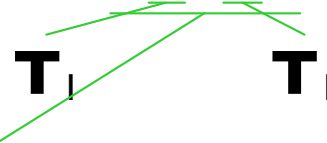
Type Inference: Function Calls



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

Type Inference: Cyclic Equations

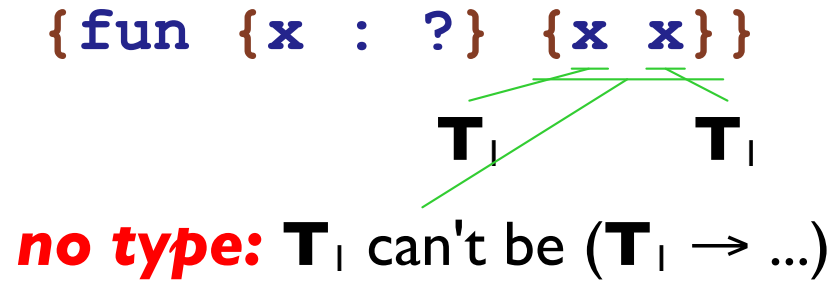
{ fun {x : ?} {x x} }



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

- \mathbf{T}_1 can't be *number*
- \mathbf{T}_1 can't be *boolean*
- 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



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 *number* or *boolean*, 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>}
```

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



Representing Expressions

```
(define-type TIFAE
  [num (n : number)]
  [add (lhs : TIFAE)
       (rhs : TIFAE)]
  [sub (lhs : TIFAE)
       (rhs : TIFAE)]
  [id (name : symbol)]
  [fun (param-name : symbol)
       (param-te : TE) ; not a type!
       (body : TIFAE)]
  [app (fun-expr : TIFAE)
       (arg-expr : TIFAE)])
```


Representing Type Variables

```
(define-type Type
  [numberT]
  [arrowT (param : Type)
          (result : Type)]
  [varT (is : (boxof MaybeType))])
```

```
(define-type TE
  [numberTE]
  [arrowTE
   (param : TE)
   (result : TE)]
  [guessTE])
```

```
(define-type MaybeType
  [none]
  [some (t : Type)])
```

Instantiating Types

```
(define instantiate-type : (TE -> Type)
  (lambda (te)
    (type-case TE te
      [numberTE () (numberT)]
      [arrowTE (d r)
               (arrowT (instantiate-type d)
                        (instantiate-type r))]
      [guessTE () (varT (box (none)))])))
```

Type Unification

```
(define unify! : (Type Type -> void)
  (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)
                  (void)
                  (set-box!
                    b
                    (some t2))))))]
            [else ...])]]])
      [numberT () ...]
      [arrowT (d1 r1) ...]))))
```

Type Unification

```
(define unify! : (Type Type -> void)
  (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 'typecheck "failed")
              (set-box! b (some t2))))))]
      [numberT () ...]
      [arrowT (d1 r1) ...])))
```

Type Unification

```
(define unify! : (Type Type -> void)
  (lambda (t1 t2)
    (type-case Type t1
      [varT (b) ...]
      [numberT () (type-case Type t2
        [varT (b) (unify! t2 t1)]
        [numberT () (void)]
        [else (error 'typecheck "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 'typecheck "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)])))]
      [numberT () false]
      [arrowT (d r) (or (occurs? b d)
                       (occurs? b r))]))))
```

TIFAE Type Checker

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

TIFAE Type Checker

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


TIFAE Type Checker

```
(define typecheck : (TIFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TIFAE fae
      ...
      [id (name) (lookup-type name env)]
      [fun (param-name param-te body)
        (local [(define param-type (instantiate-type param-te))]
          (arrowT param-type
            (typecheck body (aBind param-name
                                   param-type
                                   env)))))]
      ...)))
```

TIFAE Type Checker

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