Type Inference

Type Declarations

$$\Gamma \vdash \texttt{<num>} : \textit{number} \qquad [\ldots\ \texttt{<id>}\leftarrow\tau\ \ldots] \vdash \texttt{<id>} : \tau$$

$$\Gamma \vdash \texttt{true} : \textit{boolean} \qquad \Gamma \vdash \texttt{false} : \textit{boolean}$$

$$\frac{\Gamma \vdash \mathbf{e}_1 : \textit{number} \qquad \Gamma \vdash \mathbf{e}_2 : \textit{number}}{\Gamma \vdash \{\texttt{+}\ \mathbf{e}_1\ \mathbf{e}_2\} : \textit{number}}$$

$$\frac{\Gamma \vdash \mathbf{e}_1 : \textit{boolean} \qquad \Gamma \vdash \mathbf{e}_2 : \tau_0 \qquad \Gamma \vdash \mathbf{e}_3 : \tau_0}{\Gamma \vdash \{\texttt{if}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3\} : \tau_0}$$

$$\frac{\Gamma[\texttt{<id>}\leftarrow\tau_1] \vdash \mathbf{e} : \tau_0}{\Gamma \vdash \{\texttt{fun}\ \{\texttt{<id>} : \tau_1\}\ \mathbf{e}\} : (\tau_1 \rightarrow \tau_0)}$$

$$\frac{\Gamma \vdash \mathbf{e}_0 : (\tau_1 \rightarrow \tau_0) \qquad \Gamma \vdash \mathbf{e}_1 : \tau_1}{\Gamma \vdash \{\mathbf{e}_0\ \mathbf{e}_1\} : \tau_0}$$

Type Inference

$$\Gamma \vdash \texttt{<num>} : \textit{number} \qquad [\ldots\ \texttt{<id>}\leftarrow\tau\ \ldots] \vdash \texttt{<id>} : \tau$$

$$\Gamma \vdash \texttt{true} : \textit{boolean} \qquad \Gamma \vdash \texttt{false} : \textit{boolean}$$

$$\frac{\Gamma \vdash \mathbf{e}_1 : \textit{number} \qquad \Gamma \vdash \mathbf{e}_2 : \textit{number}}{\Gamma \vdash \{\texttt{+}\ \mathbf{e}_1\ \mathbf{e}_2\} : \textit{number}}$$

$$\frac{\Gamma \vdash \mathbf{e}_1 : \textit{boolean} \qquad \Gamma \vdash \mathbf{e}_2 : \tau_0 \qquad \Gamma \vdash \mathbf{e}_3 : \tau_0}{\Gamma \vdash \{\texttt{if}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3\} : \tau_0}$$

$$\frac{\Gamma[\texttt{<id>}\leftarrow\tau_1] \vdash \mathbf{e} : \tau_0}{\Gamma \vdash \{\texttt{fun}\ \{\texttt{<id>}\}\ \ \mathbf{e}\} : (\tau_1 \rightarrow \tau_0)}$$

$$\frac{\Gamma \vdash \mathbf{e}_0 : (\tau_1 \rightarrow \tau_0) \qquad \Gamma \vdash \mathbf{e}_1 : \tau_1}{\Gamma \vdash \{\mathbf{e}_0\ \mathbf{e}_1\} : \tau_0}$$

Do the rules still work? (Yes.)

$$\frac{\dfrac{\texttt{[x} \leftarrow \textit{number}\texttt{]}\ \texttt{x} : \textit{number} \qquad \texttt{[x} \leftarrow \textit{number}\texttt{]}\ \texttt{1} : \textit{number}}{\texttt{[x} \leftarrow \textit{number}\texttt{]}\ \texttt{\{+ x 1\}} : \textit{number}}}{\varnothing \vdash \texttt{\{fun \{x\} \{+ x 1\}\}} : \texttt{(}\textit{number}\ \texttt{->}\ \textit{number}\texttt{)}}$$

But how do we implement them now?

Type Inference

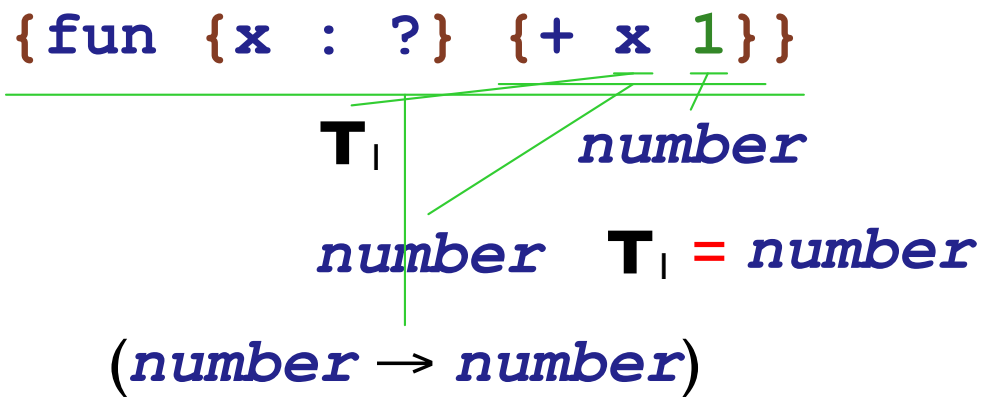
- ***Type inference*** is the process of inserting type annotations where the programmer omits them

- We'll use explicit question marks, to make it clear where types are omitted

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

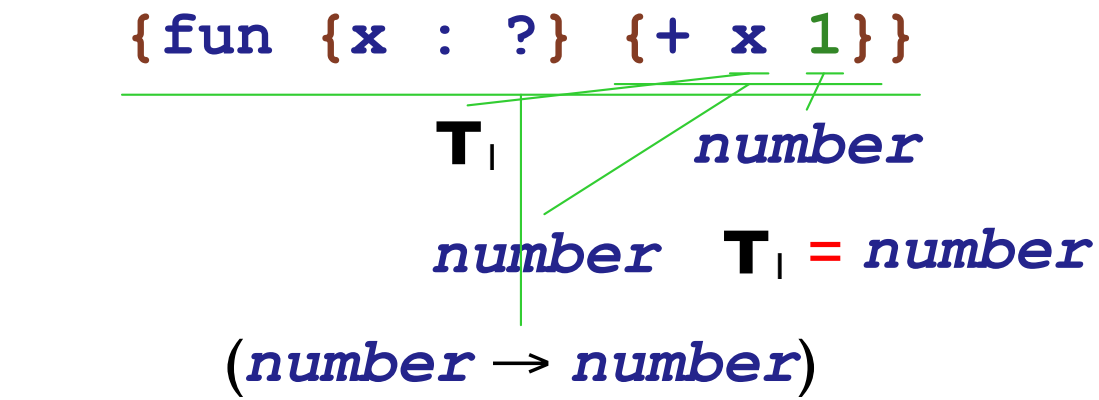
```
<TE>  ::=  number
        |  boolean
        |  (<TE> -> <TE>)
        |  ?
```

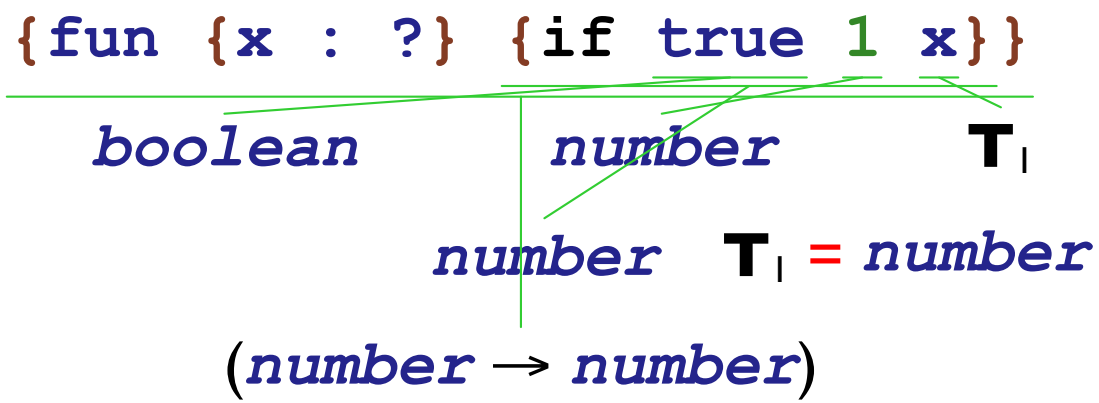
Type Inference



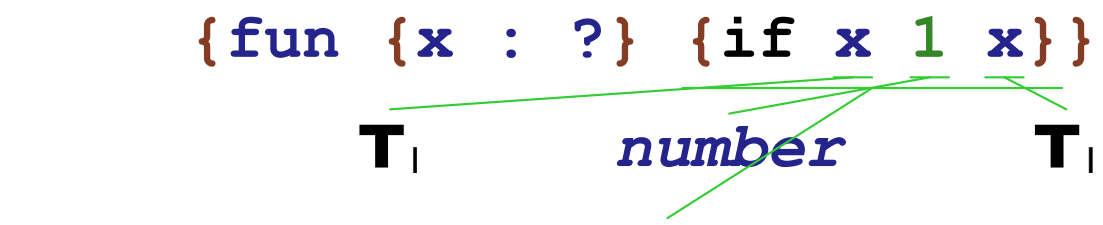
- Create a new type variable for each `?`
- Change type comparison to install type equivalences

Type Inference



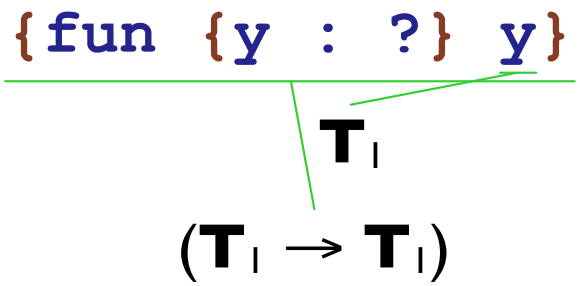


Type Inference: Impossible Cases



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

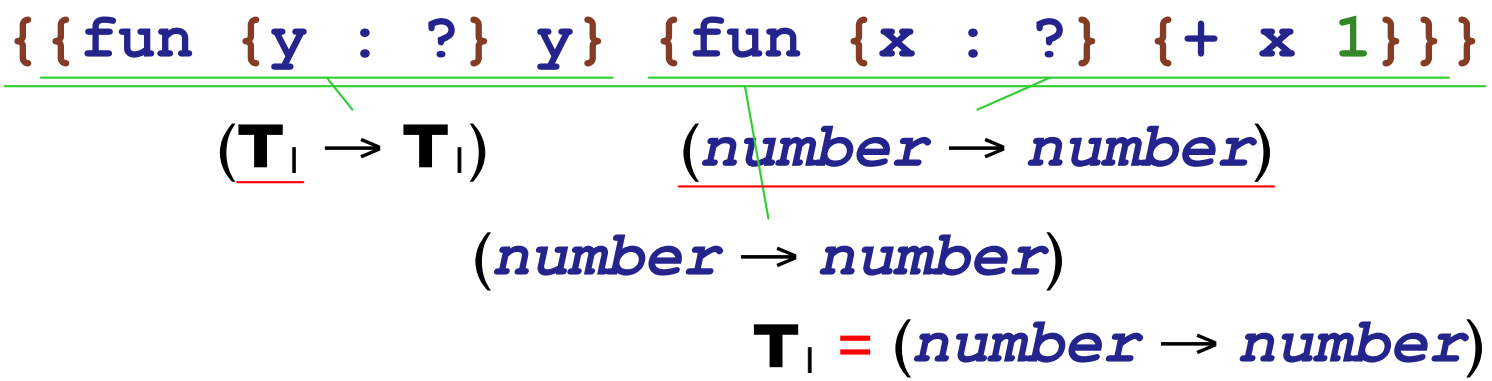
Type Inference: Many Cases



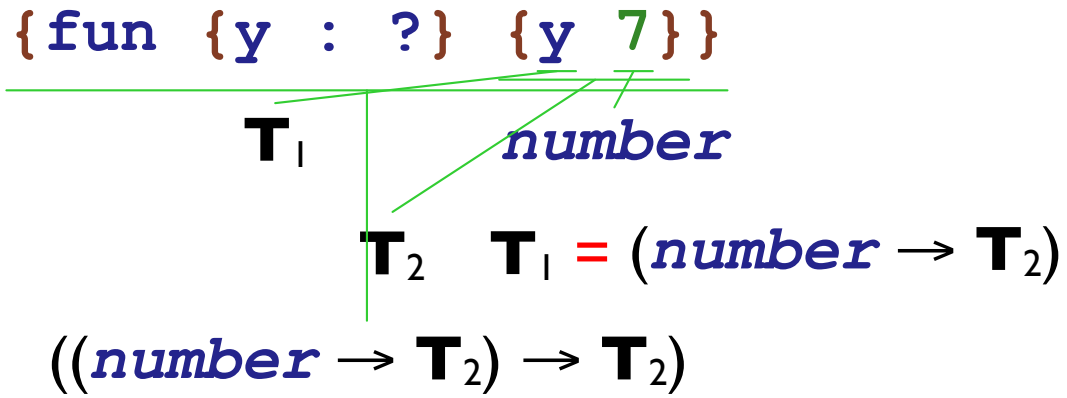
- Sometimes, more than one type works
  - (*number* → *number*)
  - (*boolean* → *boolean*)
  - ((*number* → *boolean*) → (*number* → *boolean*))

  so the type checker leaves variables in the reported type

Type Inference: Function Calls

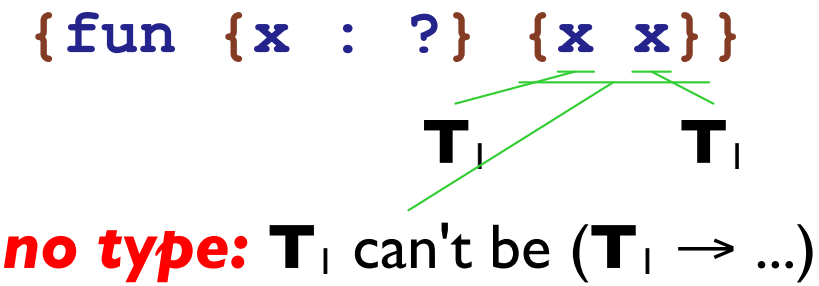


Type Inference: Function Calls



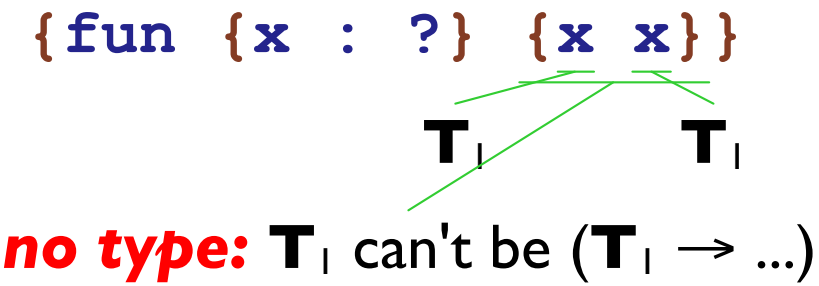
- In general, create a new type variable record for the result of a function call

Type Inference: Cyclic Equations



- $\mathbf{T}_1$ 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 **T** doesn't already contain **T**

Type Unification

Unify a type variable **T** with a type τ_2:

- If **T** is set to τ_1, unify τ_1 and τ_2
- If τ_2 is already equivalent to **T**, succeed
- If τ_2 contains **T**, then fail
- Otherwise, set **T** to τ_2 and succeed

Unify a type τ_1 to type τ_2:

- If τ_2 is a type variable **T**, then unify **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)
                                (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) (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)
           (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)
           (define result-type (varT (box (none))))
           (unify! (arrowT (typecheck arg-expr env)
                           result-type)
                   (typecheck fun-expr env))
           result-type]
      ...)))
```