Part I

Expressions and Types

What is the type of the following expression?

```
{lambda {x} {+ x 1}}
```

Answer: Yet another trick question; it's not an expression in our typed language, because the argument type is missing

But it seems like the answer *should* be (*num* → *num*)

Type Inference

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

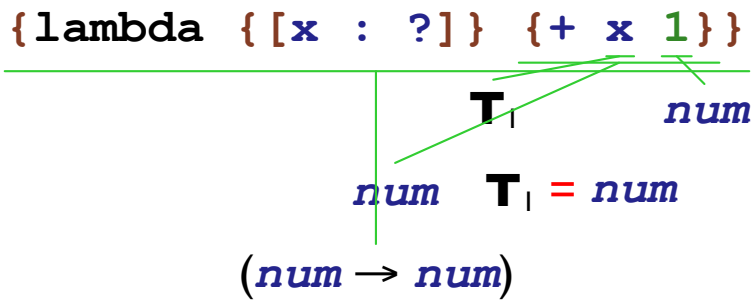
We'll use explicit question marks, to make it clear where types are omitted

```
{lambda {[x : ?]} {+ x 1}}
```

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

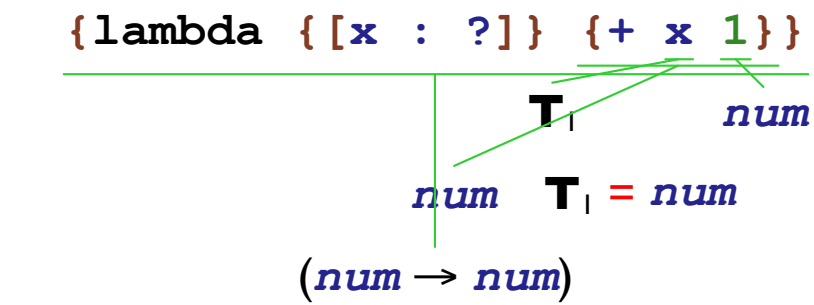
Part 2

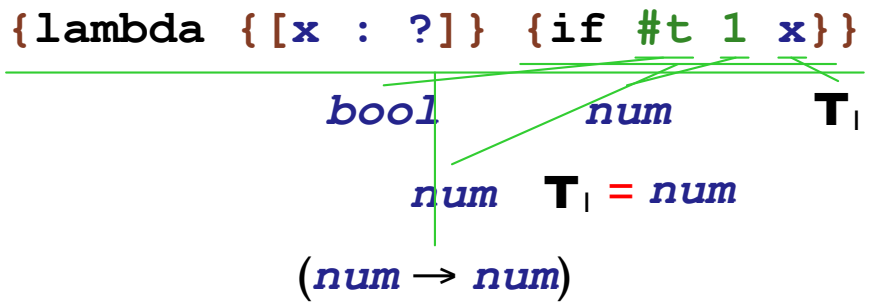
Type Inference



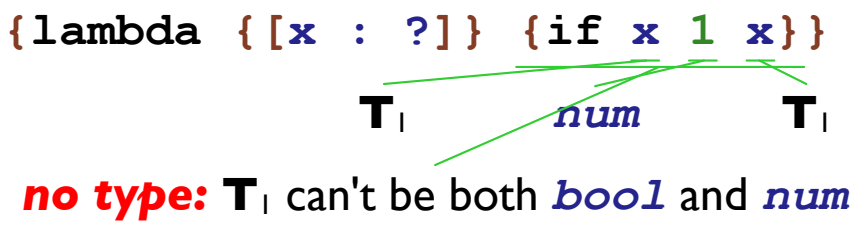
- Create a new type variable for each `?`
- Change type comparison to install type equivalences

Type Inference



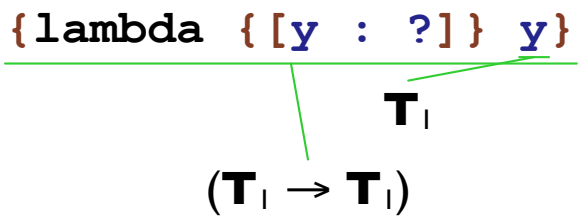


Type Inference: Impossible Cases



no type: $\mathbf{T}_1$ can't be both ***bool*** and ***num***

Type Inference: Many Cases

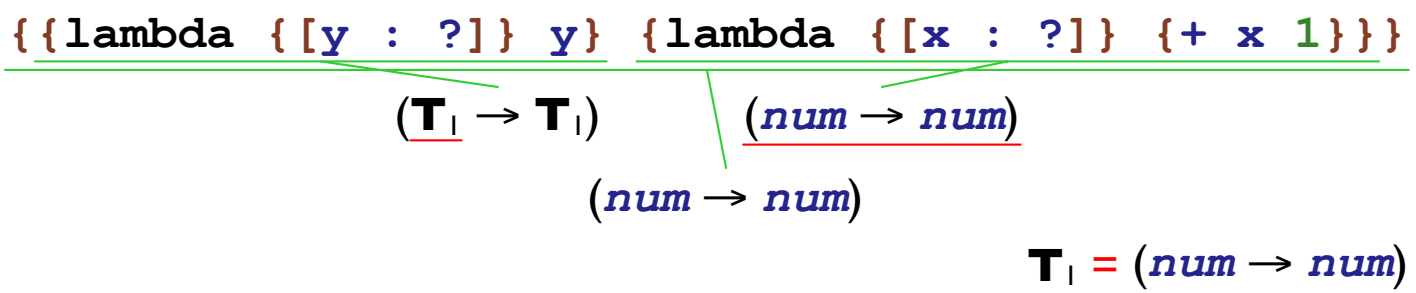


- Sometimes, more than one type works
  - (*num* → *num*)
  - (*bool* → *bool*)
  - ((*num* → *bool*) → (*num* → *bool*))

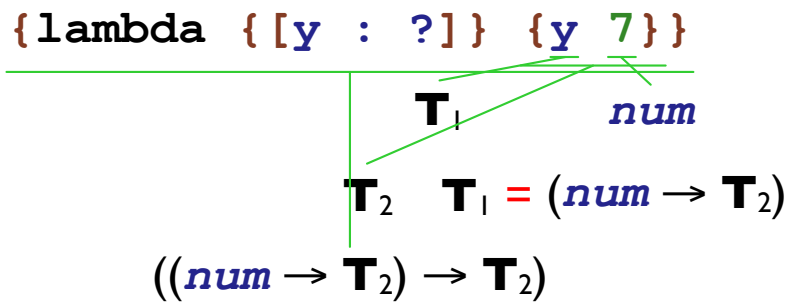
  so the type checker leaves variables in the reported type

Part 3

Type Inference: Function Calls



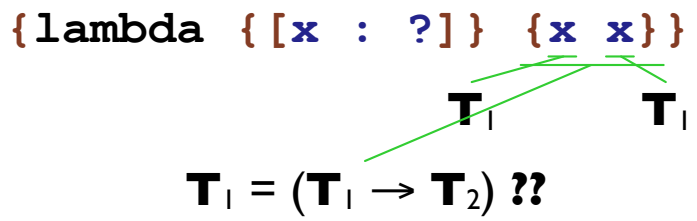
Type Inference: Function Calls



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

Part 4

Type Inference: Cyclic Equations



$\mathbf{T}_1 = (\mathbf{T}_1 \rightarrow \mathbf{T}_2) = ((\mathbf{T}_1 \rightarrow \mathbf{T}_2) \rightarrow \mathbf{T}_2)$ ... no solution

The ***occurs check***:

- When installing a type equivalence, make sure that the new type for **T** doesn't already contain **T**

Part 5

Type Unification

For comparing types, replace

```
equal? : (Type Type -> Boolean)
```

with

```
unify! : (Type Type -> ())
```

Type Unification

For comparing types, replace

```
equal? : (Type Type -> Boolean)
```

with

```
unify! : (Type Type Exp -> ())
```

To simplify by substituting with discovered equivalences:

```
resolve : (Type -> Type)
```

Type Unification

- **`resolve`** $\mathbf{T}_1 \Rightarrow \mathbf{T}_1$

- **`unify!`** $\mathbf{T}_1$ with **`num`**

  Then, **`resolve`** of $\mathbf{T}_1$ = **`num`**

- So far, **`resolve`** of $(\mathbf{T}_1 \rightarrow \mathbf{T}_2)$ = $(\texttt{num} \rightarrow \mathbf{T}_2)$

  `unify!` $\mathbf{T}_1$ with $\mathbf{T}_2$

  Then, **`resolve`** of $\mathbf{T}_2$ = **`num`**

Part 6

Type Grammar, Again

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

Representing Type Variables

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


(varT (box (none)))
```

Representing Type Variables

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


(varT (box (some (numT))))
```

Representing Type Variables

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


(define (unify! [t1 : Type] [t2 : Type] [expr : Exp])
  ....
  (type-case Type t1
    ...
    [(varT b)
     .... (set-box! b (some (resolve t2))) ....]
    ...)
  ....)
```

Part 7

Unification Examples

```
(test (unify! (numT)
              (numT))
      (values))
```

Unification Examples

```
(test (unify! (boolT)
              (boolT))
      (values))
```

Unification Examples

```
(test/exn (unify! (numT)
                  (boolT))
          "no type")
```

Unification Examples

```
(test (unify! (varT (box (none)))
              (numT))
      (values))
```

Unification Examples

```
(test (unify! (varT (box (some (numT))))
              (numT))
      (values))
```

Unification Examples

```
(test/exn (unify! (varT (box (some (boolT))))
                  (numT))
          "no type")
```

Unification Examples

```
(test/exn (let ([t (varT (box (none)))])
            (begin
              (unify! t
                      (numT))
              (unify! t
                      (boolT))))
          "no type")
```

Unification Examples

```
(test (let ([t (varT (box (none)))])
        (begin
          (unify! t
                  (numT))
          (unify! t
                  (numT))))
      (values))
```

Unification Examples

```
(test (let ([t (varT (box (none)))])
        (begin
          (unify! (arrowT t (boolT))
                  (arrowT (numT) (boolT)))
          (unify! t
                  (numT))))
      (values))
```

Unification Examples

```
(test/exn (let ([t (varT (box (none)))])
            (unify! (arrowT t (boolT))
                    t))
          "no type")
```

Unification Examples

```
(test (let ([t1 (varT (box (none)))]
            [t2 (varT (box (none)))])
        (unify! t1
                t2))
      (values))
```

Unification Examples

```
(test/exn (let ([t1 (varT (box (none)))]
                [t2 (varT (box (none)))])
            (begin
              (unify! t1
                      t2)
              (unify! t1
                      (numT))
              (unify! t2
                      (boolT))))
          "no type")
```

Unification Examples

```
(test/exn (let ([t1 (varT (box (none)))]
                [t2 (varT (box (none)))])
            (begin
              (unify! t1
                      t2)
              (unify! t2
                      (boolT))
              (unify! t1
                      (numT))))
          "no type")
```

Unification Examples

```
(test/exn (let ([t1 (varT (box (none)))]
                [t2 (varT (box (none)))])
            (begin
              (unify! t1
                      (arrowT t2 (boolT)))
              (unify! t1
                      (arrowT (numT) t2))))
          "no type")
```

Part 8

Type Unification

`unify!` a type variable **T** with a type τ_2:

- If **T** is set to τ_1, **`unify!`** τ_1 with 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

`(resolve` τ_2`)` is **T**?

`(occurs? T (resolve` τ_2`))`

`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 ***`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

Part 9

Type Unification

```
(define (unify! [t1 : Type] [t2 : Type] [expr : Exp])
  (type-case Type t1
    [(varT is1)
     ...]
    [else
     (type-case Type t2
       [(varT is2) (unify! t2 t1 expr)]
       [(numT) (type-case Type t1
                 [(numT) (values)]
                 [else (type-error expr t1 t2)])]
       [(boolT) (type-case Type t1
                  [(boolT) (values)]
                  [else (type-error expr t1 t2)])]
       [(arrowT a2 b2) (type-case Type t1
                         [(arrowT a1 b1)
                          (begin
                            (unify! a1 a2 expr)
                            (unify! b1 b2 expr))]
                         [else (type-error expr t1 t2)])])]))
```

Type Unification

```
(define (unify! [t1 : Type] [t2 : Type] [expr : Exp])
  (type-case Type t1
    [(varT is1) (type-case (Optionof Type) (unbox is1)
                  [(some t3) (unify! t3 t2 expr)]
                  [(none) (local [(define t3 (resolve t2))]
                            (if (eq? t1 t3)
                                (values)
                                (if (occurs? t1 t3)
                                    (type-error expr t1 t3)
                                    (begin
                                      (set-box! is1 (some t3))
                                      (values)))))])]
    [else ...]))
```

Type Unification Helpers

```
(define (resolve [t : Type]) : Type
  (type-case Type t
    [(varT is)
     (type-case (Optionof Type) (unbox is)
       [(none) t]
       [(some t2) (resolve t2)])]
    [else t]))

(define (occurs? [r : Type] [t : Type]) : Boolean
  (type-case Type t
    [(numT) #f]
    [(boolT) #f]
    [(arrowT a b)
     (or (occurs? r a)
         (occurs? r b))]
    [(varT is) (or (eq? r t)
                   (type-case (Optionof Type) (unbox is)
                     [(none) #f]
                     [(some t2) (occurs? r t2)]))]))
```

Part 10

Type Checker with Inference

```
(define typecheck : (Exp Type-Env -> Type)
  (lambda (a tenv)
    (type-case Exp a
      ...
      [(numE n) (numT)]
      ...)))
```

Type Checker with Inference

```
(define typecheck : (Exp Type-Env -> Type)
  (lambda (a tenv)
    (type-case Exp a
      ...
      [(plusE l r)
       (begin
         (unify! (typecheck l env) (numT) l)
         (unify! (typecheck r env) (numT) r)
         (numT))]
      ...)))
```

Type Checker with Inference

```
(define typecheck : (Exp Type-Env -> Type)
  (lambda (a tenv)
    (type-case Exp a
      ...
      [(idE name) (get-type name env)]
      [(lamE n arg-type body)
       (arrowT arg-type
               (typecheck body (aBind name
                                      arg-type
                                      env)))]
      ...)))
```

Type Checker with Inference

```
(define typecheck : (Exp Type-Env -> Type)
  (lambda (a tenv)
    (type-case Exp a
      ...
      [(appE fn arg)
       (local [(define result-type (varT (box (none))))]
         (begin
           (unify! (arrowT (typecheck arg env)
                           result-type)
                   (typecheck fn env)
                   fn)
           result-type))]
      ...)))
```

Part II

Type Errors

Checking — report that an expression doesn't have an expected type (expressed as a string):

```
type-error : (Exp String -> ...)
```

Inference — report that, near some expression, two types are incompatible:

```
type-error : (Exp Type Type -> ...)
```