Part I

Expressions and Types

What is the type of the following expression?

```
fun (x): x + 1
```

Answer: It’s not an expression in our typed variant of Moe, because the argument type is missing

But it seems like the answer *should* be `Int -> Int`

Type Inference

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

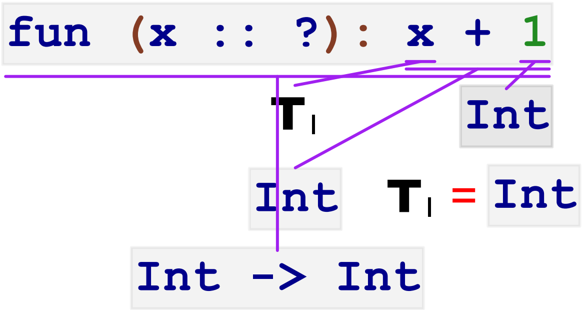
We’ll use explicit question marks, to make it clear where types are omitted

```
fun (x :: ?): x + 1
```

```
<Type>  ::=  Int
          |  Boolean
          |  <Type> -> <Type>
          |  ?
          |  (<TYPE>)
```

Part 2

Type Inference



```
fun (x :: ?): x + 1
```

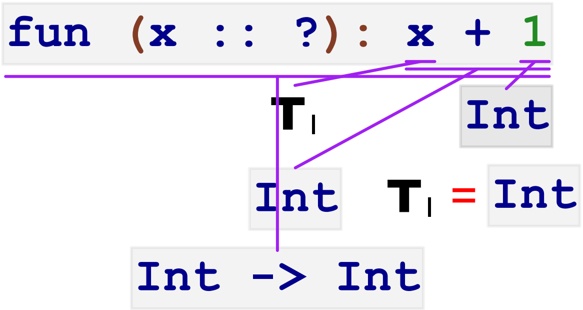
T_1 Int

Int T_1 = Int

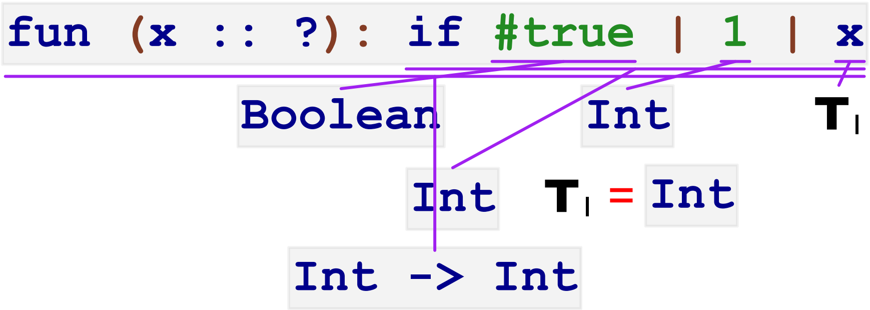
```
Int -> Int
```

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

Type Inference

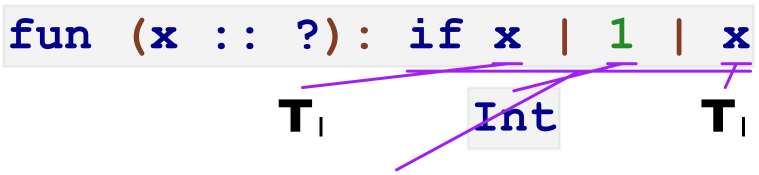


```
fun (x :: ?): x + 1
```



```
fun (x :: ?): if #true | 1 | x
```

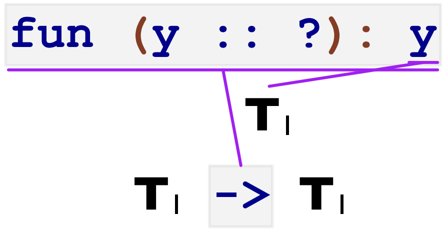
Type Inference: Impossible Cases



```
fun (x :: ?): if x | 1 | x
```

no type: **T**$_1$ can't be both `Boolean` and `Int`

Type Inference: Many Cases



```
fun (y :: ?): y
```

T_1

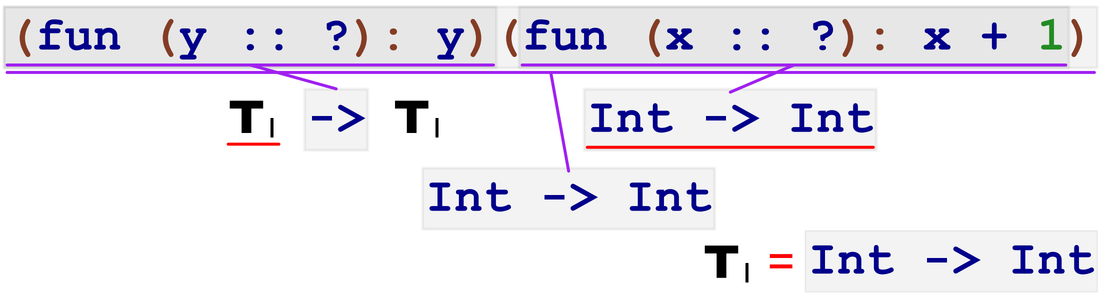
T_1 `->` T_1

- Sometimes, more than one type works
  - ○ `Int -> Int`
  - ○ `Boolean -> Boolean`
  - ○ `(Int -> Boolean) -> Int -> Boolean`

  so the type checker leaves variables in the reported type

Part 3

Type Inference: Function Calls



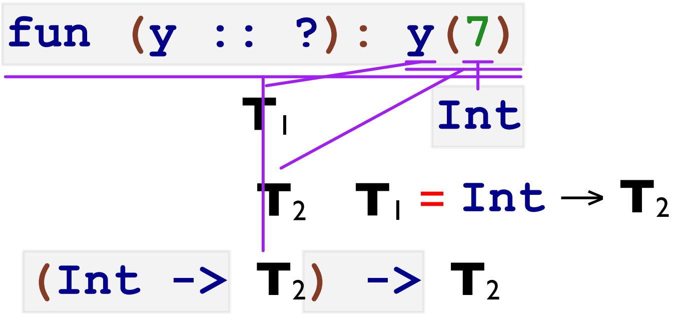
```
(fun (y :: ?) : y)(fun (x :: ?) : x + 1)
```

T_1 `->` T_1 `Int -> Int`

`Int -> Int`

T_1 = `Int -> Int`

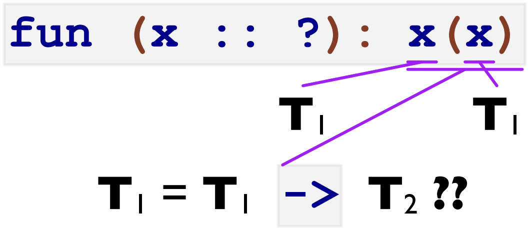
Type Inference: Function Calls



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

Part 4

Type Inference: Cyclic Equations



```
fun (x :: ?): x(x)
```

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

$\mathbf{T}_1 = \mathbf{T}_1 \; \texttt{->} \; \mathbf{T}_2$??

$\mathbf{T}_1 = \mathbf{T}_1 \; \texttt{->} \; \mathbf{T}_2 = (\mathbf{T}_1 \; \texttt{->} \; \mathbf{T}_2) \; \texttt{->} \; \mathbf{T}_2$ ... 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

```
== :: (Type, Type) -> Boolean
```

with

```
unify :: (Type, Type) -> Void
```

Type Unification

For comparing types, replace

```
== :: (Type, Type) -> Boolean
```

with

```
unify :: (Type, Type, Exp) -> Void
```

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 `Int`

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

- So far, `resolve` of $\mathbf{T}_1$ `->` $\mathbf{T}_2$ = `Int` `->` $\mathbf{T}_2$

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

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

Part 6

Type Grammar, Again

```
<Type>  ::=  Int
          |  Boolean
          |  <Type> -> <Type>
          |  ?
          |  (<TYPE>)
```

Representing Type Variables

```
type Type
| intT()
| boolT()
| arrowT(arg :: Type,
         result :: Type)
| varT(is :: Boxof(Optionof(Type)))
```

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

Representing Type Variables

```
type Type
| intT()
| boolT()
| arrowT(arg :: Type,
         result :: Type)
| varT(is :: Boxof(Optionof(Type)))


varT(box(some(intT())))
```

Representing Type Variables

```
type Type
| intT()
| boolT()
| arrowT(arg :: Type,
         result :: Type)
| varT(is :: Boxof(Optionof(Type)))


fun unify(t1 :: Type, t2 :: Type, expr :: Exp):
  .....
  match t1
  | ....
  | varT(b):
      .... set_box(b, some(resolve(t2))) ....
  | ....
  ....
```

Part 7

Unification Examples

```
check: unify(intT(),
             intT())
       ~is #void
```

Unification Examples

```
check: unify(boolT(),
             boolT())
       ~is #void
```

Unification Examples

```
check: unify(intT(),
             boolT())
       ~raises "no type"
```

Unification Examples

```
check: unify(varT(box(none())),
             intT())
       ~is #void
```

Unification Examples

```
check: unify(varT(box(some(intT()))),
             intT())
       ~is #void
```

Unification Examples

```
check: unify(varT(box(some(boolT()))),
             intT())
       ~raises "no type"
```

Unification Examples

```
check: block:
         def t = varT(box(none()))
         unify(t,
               intT())
         unify(t,
               boolT())
       ~raises "no type"
```

Unification Examples

```
check: block:
         def t = varT(box(none()))
         unify(t,
               intT())
         unify(t,
               intT())
       ~is #void
```

Unification Examples

```
check: block:
         def t = varT(box(none()))
         unify(arrowT(t, boolT()),
               arrowT(intT(), boolT()))
         unify(t,
               intT())
       ~is #void
```

Unification Examples

```
check: block:
         def t = varT(box(none()))
         unify(arrowT(t, boolT()),
               t)
       ~raises "no type"
```

Unification Examples

```
check: block:
         def t1 = varT(box(none()))
         def t2 = varT(box(none()))
         unify(t1,
               t2)
       ~is #void
```

Unification Examples

```
check: block:
         def t1 = varT(box(none()))
         def t2 = varT(box(none()))
         unify(t1,
               t2)
         unify(t1,
               intT())
         unify(t2,
               boolT())
       ~raises "no type"
```

Unification Examples

```
check: block:
         def t1 = varT(box(none()))
         def t2 = varT(box(none()))
         unify(t1,
               t2)
         unify(t2,
               boolT())
         unify(t1,
               intT())
       ~raises "no type"
```

Unification Examples

```
check: block:
         def t1 = varT(box(none()))
         def t2 = varT(box(none()))
         unify(t1,
               arrowT(t2, boolT()))
         unify(t1,
               arrowT(intT(), t2))
       ~raises "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

`resolve(`τ_2`)` is **T**?

- If τ_2 is already equivalent to **T**, succeed
- If τ_2 contains **T**, then fail

`occurs(T, resolve(`τ_2`))`

- 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 `Int` 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

Part 9

Type Unification

```
fun unify(t1 :: Type, t2 :: Type, expr :: Exp):
  match t1
  | varT(is1):
      ....
  | ~else:
      match t2
      | varT(is2): unify(t2, t1, expr)
      | intT(): match t1
                | intT(): #void
                | ~else: type_error(expr, t1, t2)
      | boolT(): match t1
                 | boolT(): #void
                 | ~else: type_error(expr, t1, t2)
      | arrowT(a2, b2): match t1
                        | arrowT(a1, b1):
                            unify(a1, a2, expr)
                            unify(b1, b2, expr)
                        | ~else: type_error(expr, t1, t2)
```

Type Unification

```
fun unify(t1 :: Type, t2 :: Type, expr :: Exp):
  match t1
  | varT(is1): match unbox(is1)
               | some(t3): unify(t3, t2, expr)
               | none(): block:
                           def t3 = resolve(t2)
                           if t1 === t3
                           | #void
                           | if occurs(t1, t3)
                             | type_error(expr, t1, t3)
                             | set_box(is1, some(t3))
  | ~else: ....
```

Type Unification Helpers

```
fun resolve(t :: Type) :: Type:
  match t
  | varT(is):
      match unbox(is)
      | none(): t
      | some(t2): resolve(t2)
  | ~else: t

fun occurs(r :: Type, t :: Type) :: Boolean:
  match t
  | intT(): #false
  | boolT(): #false
  | arrowT(a, b):
      occurs(r, a) || occurs(r, b)
  | varT(is): (r === t) || (match unbox(is)
                            | none(): #false
                            | some(t2): occurs(r, t2))
```

Part 10

Type Checker with Inference

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ...
    | intE(n): intT()
    | ...
```

Type Checker with Inference

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ...
    | plusE(l, r):
        unify(typecheck(l, env), intT(), l)
        unify(typecheck(r, env), intT(), r)
        intT()
    | ...
```

Type Checker with Inference

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ...
    | idE(name): get_type(name, env)
    | funE(n, arg_type, body):
        arrowT(arg_type,
               typecheck(body, aBind(name,
                                     arg_type,
                                     env)))
    | ...
```

Type Checker with Inference

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ...
    | appE(fn, arg):
        def result_type = varT(box(none()))
        unify(arrowT(typecheck(arg, env),
                     result_type),
              typecheck(fn, env),
              fn)
        result_type
    | ...
```

Part 11

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) -> ....
```