Type Soundness

Type soundness is a theorem of the form

> If $\varnothing \vdash \mathbf{e} : \tau$, then running $\mathbf{e}$ never produces an error

If we add division, then divide-by-zero errors may be ok:

> If $\varnothing \vdash \mathbf{e} : \tau$, then running $\mathbf{e}$ never produces an error except divide-by-zero

In general, soundness rules out a certain class of run-time errors

Soundness fails $\Rightarrow$ bug in type rules

Type Soundness in TRCFAE

TRCFAE has a bug:

```
{rec {f : (num -> num)
        f}
     {f 10}}
```

One solution: adjust the soundness theorem to allow a run-time error

Another solution: change the grammar for `rec`

```
<TIFAE> ::= ...
          | {rec {<id> : <tyexp>
                 {fun {<id> : <tyexp>}
                      <TIFAE>}}
                <TIFAE>}
```

Type Soundness in TVRCFAE

TCRCFAE has a bug, too:

```
{{withtype {foo {a num} {b num}}
           {fun {x : foo} {+ {cases foo x
                              {a {n} n}
                              {b {n} n}}}}}
 {withtype {foo {c (num -> num)} {d num}}
           {c {fun {y : num} y}}}}
```

Solution 1: no local type declarations

Solution 2: don't let **<tyid>** escape **withtype**

$$\Gamma' = \Gamma[\ \texttt{<tyid>} = \texttt{<id>}_1@\tau_1 + \texttt{<id>}_2@\tau_2,\ \texttt{<id>}_1 \leftarrow (\tau_1 \rightarrow \texttt{<tyid>}),\ \texttt{<id>}_2 \leftarrow (\tau_2 \rightarrow \texttt{<tyid>})\ ]$$

<tyid> not in τ_0

$$\frac{\Gamma' \vdash \tau_1 \qquad \Gamma' \vdash \tau_2 \qquad \Gamma' \vdash \texttt{e} : \tau_0}{\Gamma \vdash \{\texttt{withtype}\ \{\texttt{<tyid>}\ \{\texttt{<id>}_1\ \ \tau_1\}\ \{\texttt{<id>}_2\ \ \tau_2\}\}\ \texttt{e}\} : \tau_0}$$

Quiz

What is the type of the following expression?

```
{fun {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

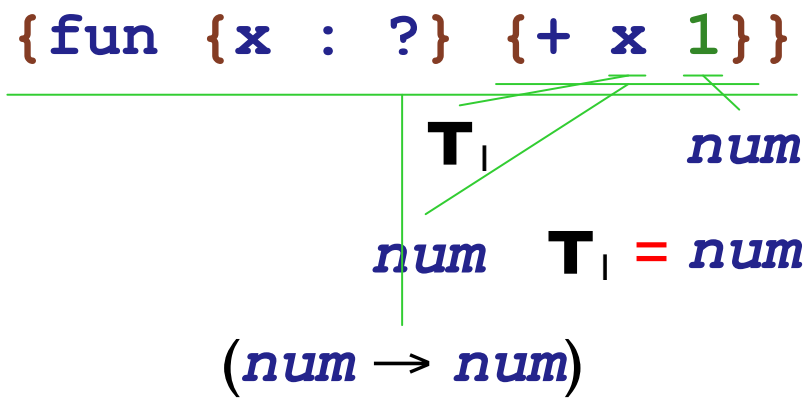
- ***Type inference*** is the process of inserting type annotations where the programmer omits them

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

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

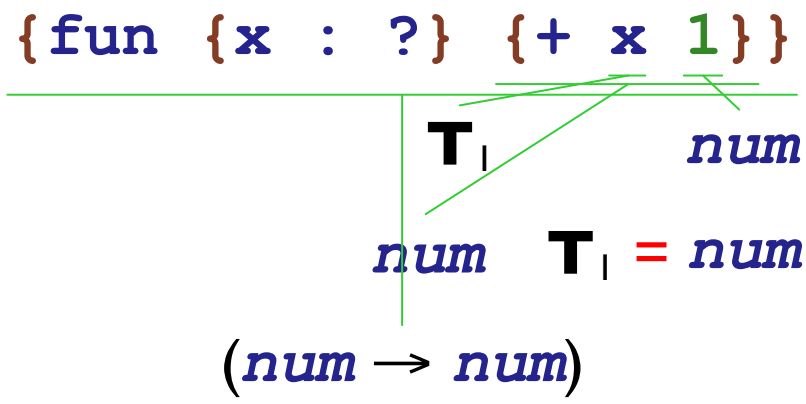
```
<typeExpr>  ::=  num
              |  bool
              |  (<typeExpr> -> <typeExpr>)
              |  ?
```

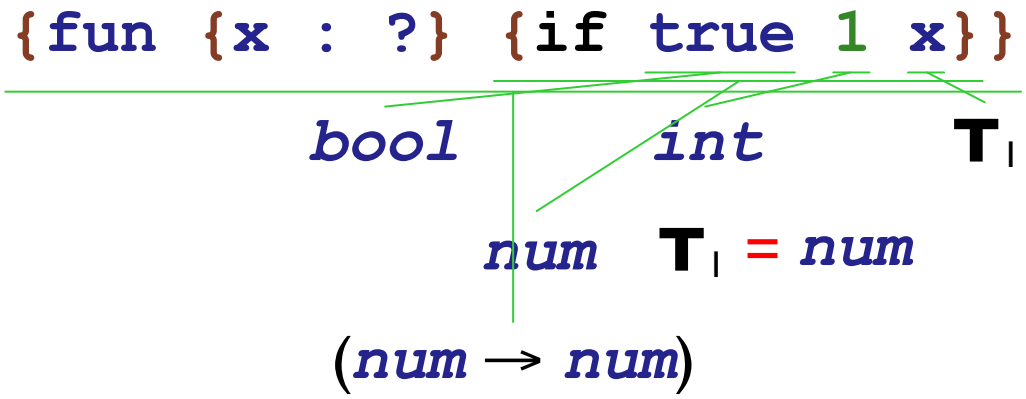
Type Inference



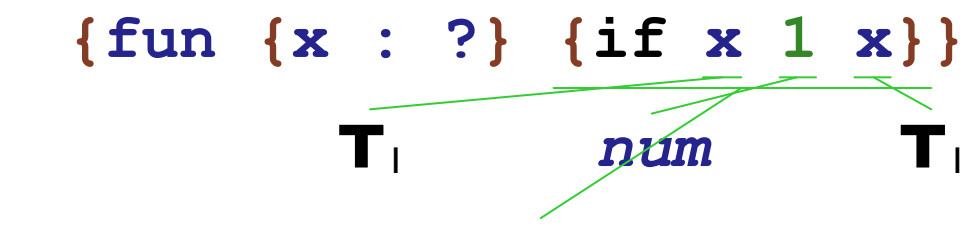
- Create a new type variable for each `?`
- Change type comparison to install type equivalences

Type Inference



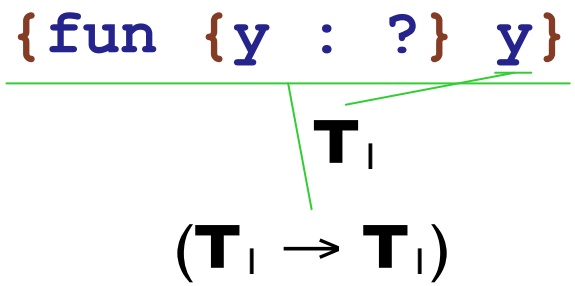


Type Inference: Impossible Cases



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

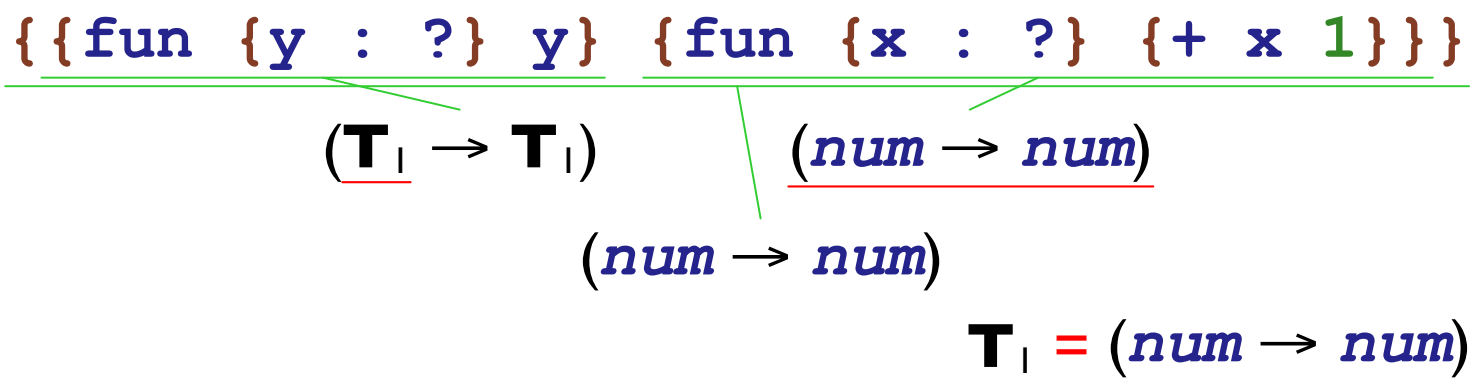
Type Inference: Many Cases



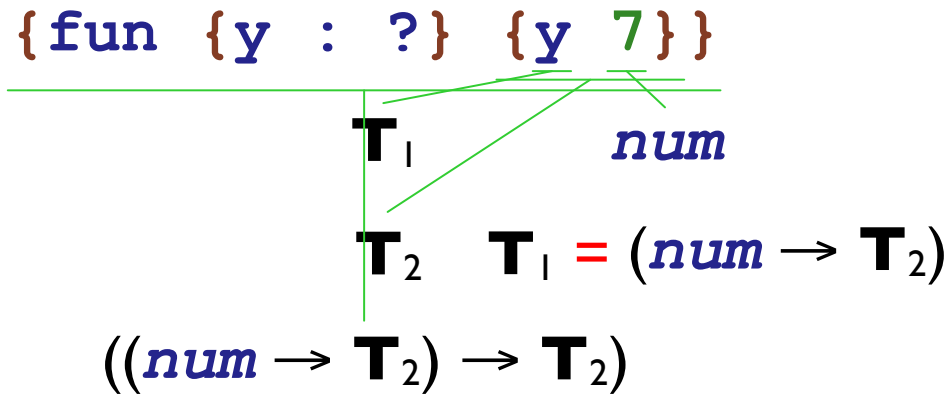
- Sometimes, more than one type works
  - (*num* → *num*)
  - (*bool* → *bool*)
  - ((*num* → *bool*) → (*num* → *bool*))

  so the type checker leaves variables in the reported type

Type Inference: Function Calls

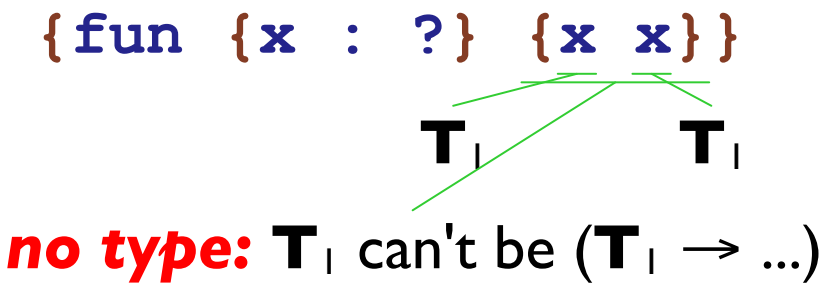


Type Inference: Function Calls



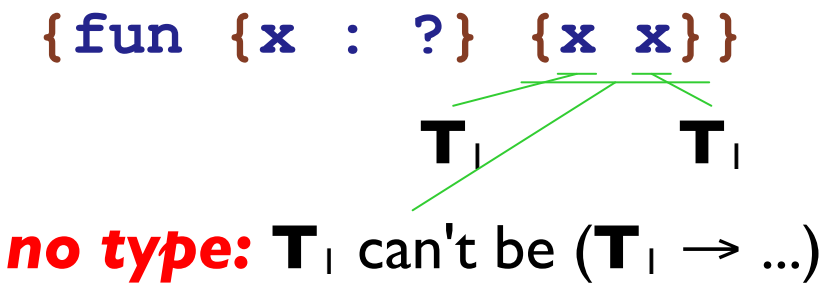
- In general, create a new type variable record for the result of a function call

Type Inference: Cyclic Equations



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



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 ***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 Type Variables

```
(define-type TE
  [numTE]
  [boolTE]
  [arrowTE (arg : TE)
           (result : TE)]
  [guessTE])

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

(define-type (Option 'alpha)
  [none]
  [some (v : 'alpha)])
```

Type Unification

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

Type Unification Helpers

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

(define (occurs? r t)
  (type-case Type t
    [numT () false]
    [boolT () false]
    [arrowT (a b)
            (or (occurs? r a)
                (occurs? r b))]
    [varT (is) (or (eq? r t)
                   (type-case (Option Type) (unbox is)
                     [none () false]
                     [some (t2) (occurs? r t2)]))]))
```

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) (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)
           (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)
           (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)))]
      ...)))
```