Type Soundness

Type soundness is a theorem of the form

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

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

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

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

Soundness fails ⇒ bug in type rules

Type Soundness in TRCFAE

TRCFAE has a bug:

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

Usual 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**

$$\frac{\begin{array}{c}\Gamma'=\Gamma[\,\texttt{<tyid>} = \texttt{<id>}_1@\tau_1+\texttt{<id>}_2@\tau_2,\ \texttt{<id>}_1\leftarrow(\tau_1 \rightarrow \texttt{<tyid>}),\ \texttt{<id>}_2\leftarrow(\tau_2 \rightarrow \texttt{<tyid>})\,]\\ \texttt{<tyid>}\text{ not in }\tau_0\\ \Gamma' \vdash \tau_1 \qquad \Gamma' \vdash \tau_2 \qquad \Gamma' \vdash \texttt{e :}\ \tau_0\end{array}}{\Gamma \vdash \texttt{\{withtype \{<tyid> \{<id>}_1\ \ \tau_1\texttt{\} \{<id>}_2\ \ \tau_2\texttt{\}\} e\}}\ :\ \tau_0}$$

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

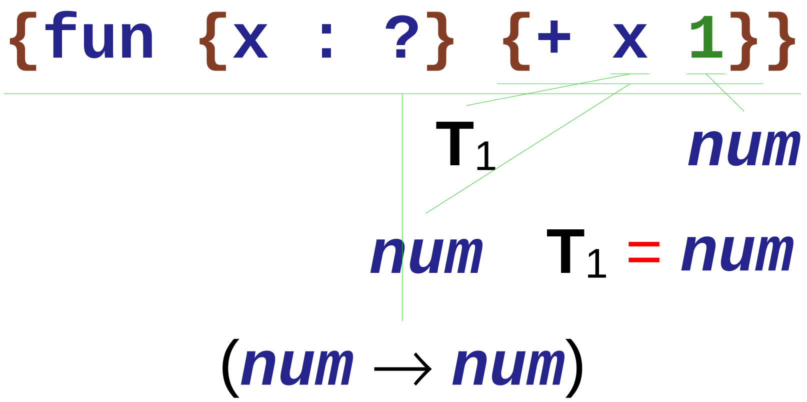
- ***Type inference*** is the process of inserting type annotations where the programmer omits them

- 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



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

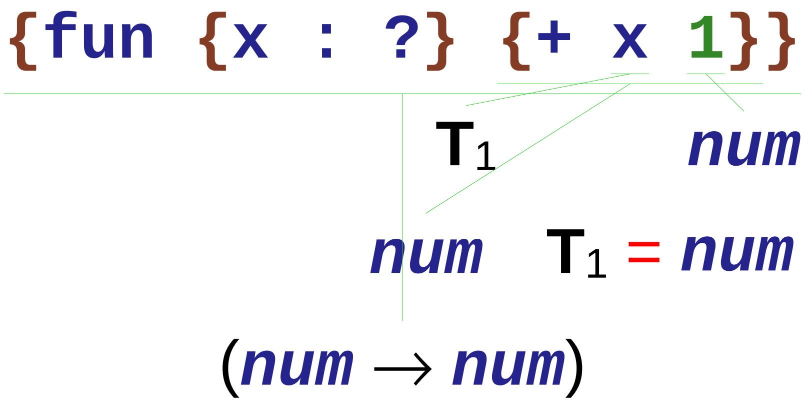
T_1 *num*

num T_1 = *num*

(*num* → *num*)

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

Type Inference



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

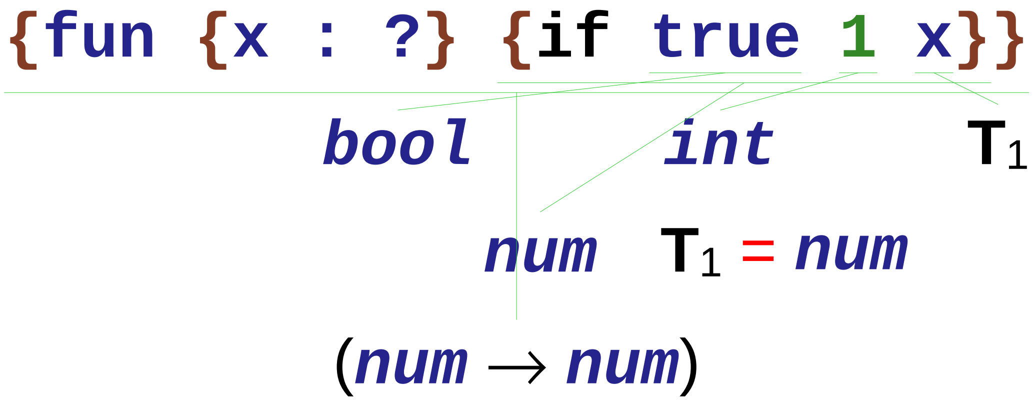
T_1

num

num

$T_1 = num$

$(num \rightarrow num)$



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

bool

int

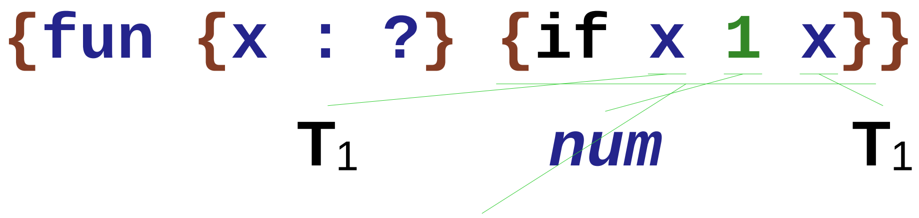
T_1

num

$T_1 = num$

$(num \rightarrow num)$

Type Inference: Impossible Cases

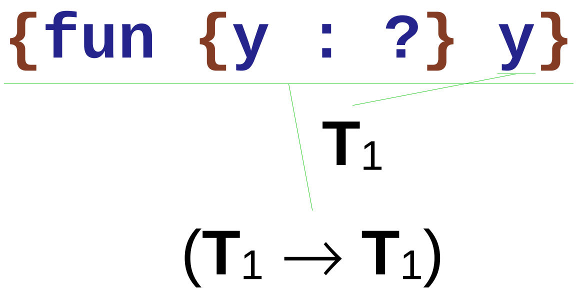


```
{fun {x : ?} {if x 1 x}}
```

T_1 *num* T_1

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

Type Inference: Many Cases



```
{fun {y : ?} y}
```

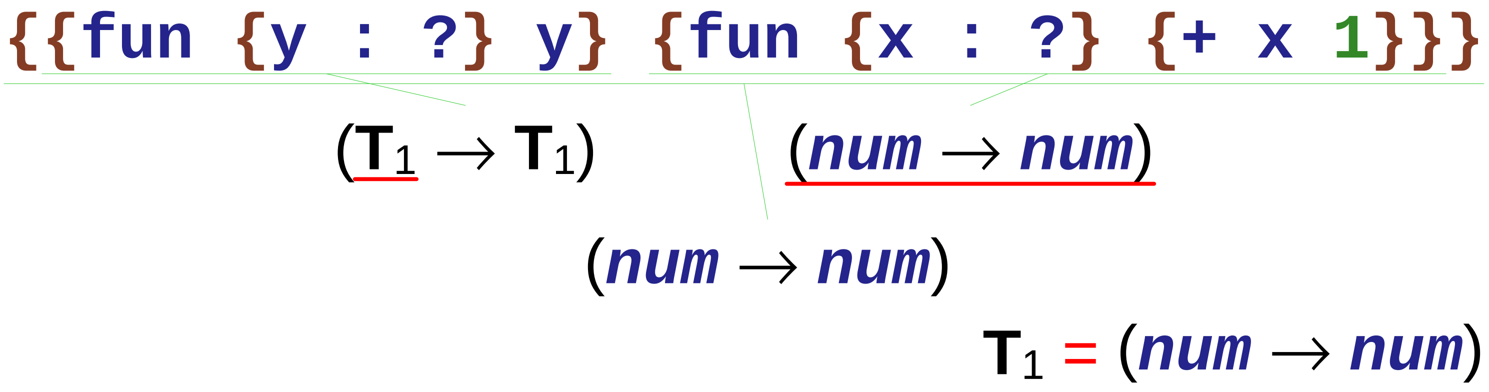
T_1

$(T_1 \rightarrow T_1)$

- 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



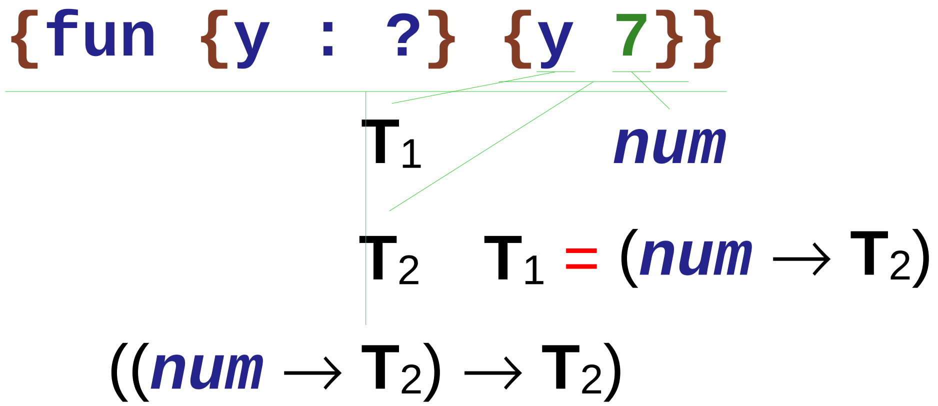
```
{{fun {y : ?} y} {fun {x : ?} {+ x 1}}}
```

$(\underline{T_1} \rightarrow T_1)$ $\underline{(num \rightarrow num)}$

$(num \rightarrow num)$

$T_1 = (num \rightarrow num)$

Type Inference: Function Calls



```
{fun {y : ?} {y 7}}
```

T_1 $\quad$ num

T_2 $\quad$ $T_1 = (\mathit{num} \rightarrow T_2)$

$((\mathit{num} \rightarrow T_2) \rightarrow T_2)$

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

Type Inference: Cyclic Equations

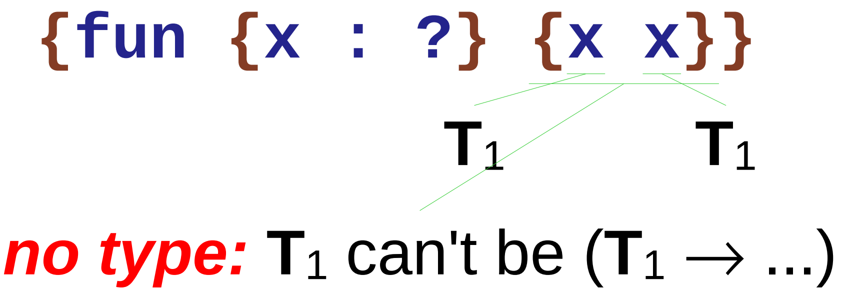
```
{fun {x : ?} {x x}}
```

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

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

- $\mathbf{T}_1$ 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



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

T_1 **T_1**

no type: **T_1** can't be (**T_1** → ...)

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> : <tyexp>} <TIFAE>}
           |  {<TIFAE> <TIFAE>}
           |  {if0 <TIFAE> <TIFAE> <TIFAE>}
           |  {rec {<id> : <tyexp> <TIFAE>} <TIFAE>}
<tyexp>  ::=  num
           |  (<tyexp> -> <tyexp>)
           |  ?
```

NEW

Representing Type Variables

```
type te =
    NumTE
  | BoolTE
  | ArrowTE of te * te
  | GuessTE

...
and ty =
    NumT
  | BoolT
  | ArrowT of ty * ty
  | VarT of ty option ref
```

Type Unification

```
let rec unify = function
    (t1, t2, expr) -> match (t1, t2) with
      (VarT(r), _) ->
        (match !r with
          Some(t1) -> unify(t1, t2, expr)
        | None ->
            let t2 = resolve(t2)
            in if samevar(t1, t2)
            then ()
            else if occurs(r, t2)
            then raise (NoType(expr, "occurs check failed"))
            else r := Some(t2))
    | (_, VarT(r)) -> unify(t2, t1, expr)
    | (ArrowT(a1, b1), ArrowT(a2, b2))->
        (unify(a1, a2, expr);
         unify(b1, b2, expr))
    | (NumT, NumT) -> ()
    | (BoolT, BoolT) -> ()
    | _ ->  raise (NoType(expr, "unification failed"))
```

Type Unification Helpers

```
let rec resolve = function
    t -> match t with
      VarT(r) ->
        (match !r with
          None -> t
        | Some(t) -> resolve(t))
    | _ -> t

let samevar = function
    (VarT(r1), VarT(r2)) -> r1 == r2
  | _ -> false

let rec occurs = function
    (r, t) -> match t with
      NumT -> false
    | BoolT -> false
    | ArrowT(a, b) -> occurs(r, a) || occurs(r, b)
    | VarT(r2) ->
        ((r == r2) ||
        (match !r2 with
          None -> false
        | Some(t) -> occurs(r, t)))
```

TIFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    (expr, env) -> match expr with
      Num(n) -> NumT
      …
```

TIFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
  (expr, env) -> match expr with
   …
  | Add(l, r) ->
    (unify(typecheck(l,env), NumT, l);
     unify(typecheck(r,env), NumT, r);
     NumT)
   …
```

TIFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    (expr, env) -> match expr with
      …
    | Sub(l, r) ->
        (unify(typecheck(l,env), NumT, l);
         unify(typecheck(r,env), NumT, r);
         NumT)
      …
```

TIFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    (expr, env) -> match expr with
      …
    | Id(name) -> gettype(name, env)
    | Fun(param, texpr, body) ->
        let argType = parseType(texpr)
        in ArrowT(argType,
                  typecheck(body, ABind(param,
                                        argType,
                                        env)))
      …
```

TIFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    (expr, env) -> match expr with
      …
    | App(fn, arg) ->
        let resultType = VarT(ref None)
        in (unify(typecheck(fn, env),
                  ArrowT(typecheck(arg, env), resultType),
                  expr);
            resultType)
      …
```

TIFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    (expr, env) -> match expr with
      …
    | IfZ(tst, thn, els) ->
        (unify(typecheck(tst, env), NumT, tst);
         let thnType = typecheck(thn, env)
         and elsType = typecheck(els, env)
         in (unify(thnType, elsType, expr);
             thnType))
      …
```

TIFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
  (expr, env) -> match expr with
   …
  | Rec(name, texpr, rhs, body) ->
    let bindType = parseType(texpr)
    in let env = ABind(name, bindType, env)
    in (unify(typecheck(rhs, env), bindType, expr);
        typecheck(body, env))
   …
```