Type Soundness

Type soundness is a theorem of the form

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

Type Soundness

Type soundness is a theorem of the form

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

If we add division and arrays, then some errors are ok:

If $\varnothing \vdash \mathbf{e} : \tau$, then running **e** never produces an error except divide-by-zero or array-out-of-bounds

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

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 and arrays, then some errors are ok:

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

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

Soundness fails $\Rightarrow$ bug in type rules or bug in eval

Quiz

What is the type of the following expression?

```
{fun {x} {+ x 1}}
```

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

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 ($num \rightarrow num$)

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}}
```

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}}
```

Type Inference

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

$\mathbf{T}_1$

Type Inference

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

T_1 *num*

Type Inference

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

T_1 *num*

num T_1 = *num*

Type Inference

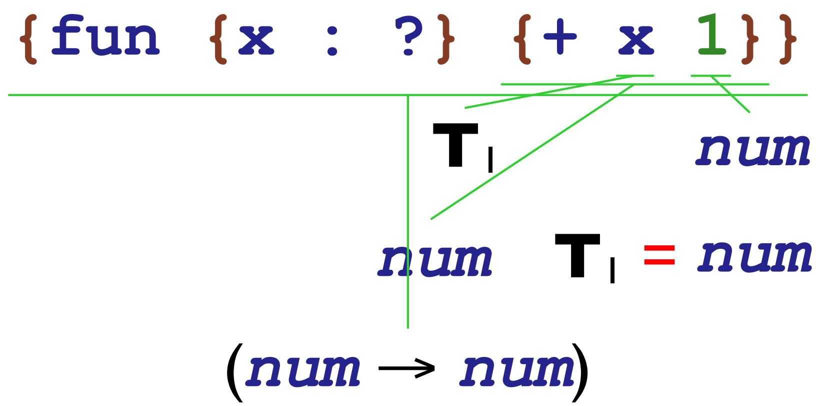
```
{fun {x : ?} {+ x 1}}
```

T_1 *num*

num T_1 = *num*

(*num* → *num*)

Type Inference



- 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* → *num*)

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

Type Inference

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

T_1 *num*

num T_1 = *num*

(*num* → *num*)

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

bool

Type Inference

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

T_1 *num*

num T_1 = *num*

(*num* → *num*)

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

bool *int*

Type Inference

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

T_1 *num*

num T_1 = *num*

(*num* → *num*)

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

bool *int* T_1

Type Inference

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

T_1 *num*

num T_1 = *num*

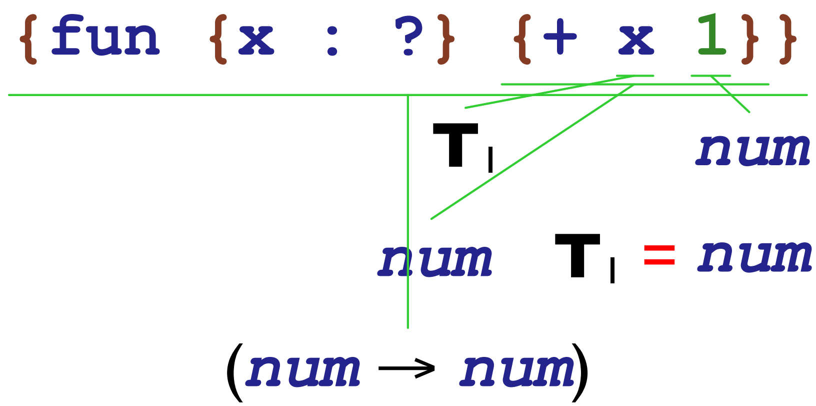
(*num* → *num*)

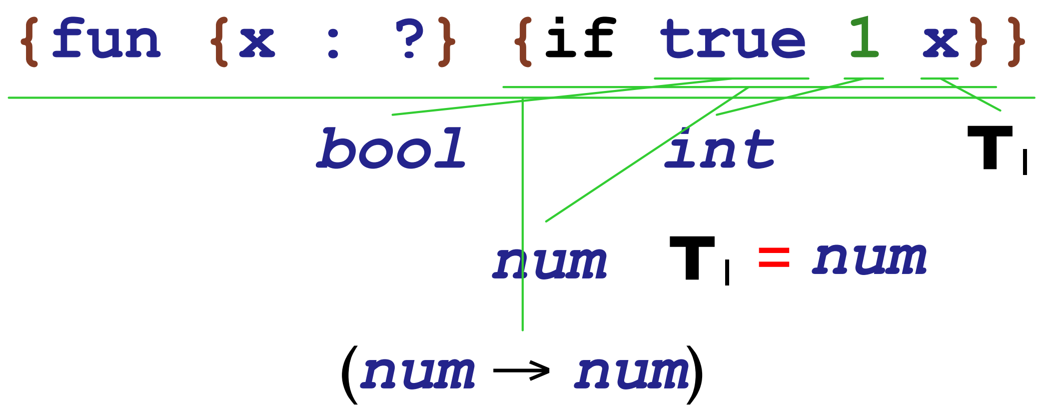
```
{fun {x : ?} {if true 1 x}}
```

bool *int* T_1

num T_1 = *num*

Type Inference





Type Inference: Impossible Cases

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

Type Inference: Impossible Cases

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

T_1

Type Inference: Impossible Cases

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

T_1 *num*

Type Inference: Impossible Cases

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

T_1 *num* T_1

Type Inference: Impossible Cases

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

$\mathbf{T}_1$ *num* $\mathbf{T}_1$

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

Type Inference: Many Cases

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

Type Inference: Many Cases

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

T_1

Type Inference: Many Cases

`{fun {y : ?} y}`

T_1

$(T_1 \rightarrow T_1)$

Type Inference: Many Cases

`{fun {y : ?} y}`

T_1

$(T_1 \rightarrow T_1)$

- Sometimes, more than one type works
  - ($\textit{num} \rightarrow \textit{num}$)
  - ($\textit{bool} \rightarrow \textit{bool}$)
  - (($\textit{num} \rightarrow \textit{bool}$) $\rightarrow$ ($\textit{num} \rightarrow \textit{bool}$))

  so the type checker leaves variables in the reported type

Type Inference: Function Calls

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

Type Inference: Function Calls

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

$(\mathbf{T}_1 \rightarrow \mathbf{T}_1)$

Type Inference: Function Calls

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

$(\mathbf{T}_1 \rightarrow \mathbf{T}_1)$ $(num \rightarrow num)$

Type Inference: Function Calls

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

$(\mathbf{T}_1 \rightarrow \mathbf{T}_1)$ $(\mathit{num} \rightarrow \mathit{num})$

$\mathbf{T}_1 = (\mathit{num} \rightarrow \mathit{num})$

Type Inference: Function Calls

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

$(T_1 \rightarrow T_1)$ $(num \rightarrow num)$

$(num \rightarrow num)$

$T_1 = (num \rightarrow num)$

Type Inference: Function Calls

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

Type Inference: Function Calls

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

T_1

Type Inference: Function Calls

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

T_1 *num*

Type Inference: Function Calls

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

T_1 *num*

T_2 $T_1 = (num \rightarrow T_2)$

Type Inference: Function Calls

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

T_1 *num*

T_2 $T_1 = (num \rightarrow T_2)$

$((num \rightarrow T_2) \rightarrow T_2)$

Type Inference: Function Calls

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

T_1 $\quad$ num

$T_2 \quad T_1 = (num \rightarrow T_2)$

$((num \rightarrow T_2) \rightarrow T_2)$

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

Type Inference: Cyclic Equations

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

Type Inference: Cyclic Equations

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

T_1

Type Inference: Cyclic Equations

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

T_1 T_1

Type Inference: Cyclic Equations

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

T_1 T_1

no type: T_1 can't be ($T_1 \rightarrow$ ...)

- T_1 can't be *`int`*
- T_1 can't be *`bool`*
- Suppose T_1 is ($T_2 \rightarrow T_3$)
  - T_2 must be T_1
  - So we won't get anywhere!

Type Inference: Cyclic Equations

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

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

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

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>)
           |  ?
```

NEW

Representing Expressions

```
(define-type TFAE
  [num (n : number)]
  [add (l : TFAE)
       (r : TFAE)]
  [sub (l : TFAE)
       (r : TFAE)]
  [id (name : symbol)]
  [fun (name : symbol)
       (t : TE)   ; different
       (body : TFAE)]
  [app (rator : TFAE)
       (rand : TFAE)])
```

Representing Type Variables

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

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

```
(define-type MaybeType
  [none]
  [some (t : Type)])
```

Parsing Types

```
(define parse-type : (TE -> Type)
  (lambda (te)
    (type-case TE te
      [numTE () (numT)]
      [arrowTE (d r)
               (arrowT (parse-type d)
                       (parse-type r))]
      [guessTE () (varT (box (none)))])))
```

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) ...]
    [else ...]))
```

Type Unification

```
(define (unify! t1 t2 expr)
  (type-case Type t1
    [varT (is1) ...]
    [else ...]))
```

Type Unification

```
(define (unify! t1 t2 expr)
  (type-case Type t1
    [varT (is1) (type-case MaybeType (unbox is1)
                  [some (t3) (unify! t3 t2 expr)]
                  [none () ...])]
    [else ...]))
```

Type Unification

```
(define (unify! t1 t2 expr)
  (type-case Type t1
    [varT (is1) (type-case MaybeType (unbox is1)
                  [some (t3) (unify! t3 t2 expr)]
                  [none () (local [(define t3 (resolve t2))]
                             ...)])]
    [else ...]))
```

Type Unification

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

Type Unification

```
(define (unify! t1 t2 expr)
  (type-case Type t1
    [varT (is1) (type-case MaybeType (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 MaybeType (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 MaybeType (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)
           ... (typecheck l env) ...
           ... (typecheck r env) ...]
      ...)))
```

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)
           ... (typecheck fn env) ...
           ... (typecheck arg 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))))]
             ... (arrowT (typecheck arg env)
                         result-type)
             ... (typecheck fn 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)
           ... (typecheck test-expr env) ...
           ... (typecheck then-expr env) ...
           ... (typecheck else-expr env) ...]
      ...)))
```

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)
             ... (typecheck then-expr env) ...
             ... (typecheck else-expr env) ...)]
      ...)))
```

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