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 $\Rightarrow$ 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`

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

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* → *num*)

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

bool *int* T_1

num T_1 = *num*

(*num* → *num*)

Type Inference: Impossible Cases

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

T$_1$ ***num*** **T**$_1$

no type: **T**$_1$ can't be both ***bool*** and ***num***

Type Inference: Many Cases

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

$\mathbf{T}_1$

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

- 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}}}
```

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

$(\textit{num} \rightarrow \textit{num})$

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

Type Inference: Function Calls

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

T_1 *num*

T_2 $T_1 = (\textit{num} \rightarrow T_2)$

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

- 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}}
```

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

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

Type Unification

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>
           |  <bool>
           |  {+ <TIFAE> <TIFAE>}
           |  {- <TIFAE> <TIFAE>}
           |  <id>
           |  {fun {<id> : <TE>} <TIFAE>}
           |  {<TIFAE> <TIFAE>}
           |  {if0 <TIFAE> <TIFAE> <TIFAE>}
           |  {rec {<id> : <TE> <TIFAE>} <TIFAE>}

<TE>     ::=  num
           |  bool
           |  (<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 (Optionof Type)))])
```

Type Unification

```
(define (unify! t1 t2 expr)
  (type-case Type t1
    [(VarT is1) (unify-type-var! is1 t2 expr)]
    [else
     (type-case Type t2
       [(VarT is2) (unify-type-var! is2 t1 expr)]
       ...)]))
```

Type Unification

```
(define (unify! t1 t2 expr)
  (type-case Type t1
    [(VarT is1) (unify-type-var! is1 t2 expr)]
    [else
     (type-case Type t2
       [(VarT is2) (unify-type-var! is2 t1 expr)]
       [(NumT) (unify-assert! t1 (NumT))]
       [(BoolT) (unify-assert! t1 (BoolT))]
       [(ArrowT a2 b2) (....)])]))
```

Type Unification

```
(define (unify! t1 t2 expr)
  (type-case Type t1
    [(VarT is1) (unify-type-var! is1 t2 expr)]
    [else
     (type-case Type t2
       [(VarT is2) (unify-type-var! is2 t1 expr)]
       [(NumT) (unify-assert! t1 (NumT) expr)]
       [(BoolT) (unify-assert! t1 (BoolT) expr)]
       [(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-type-var! T tau2 expr)
  (type-case (Optionof Type) (unbox T)
    [(some t3) (unify! t3 tau2 expr)]
    [(none)
     (let ([t3 (resolve tau2)]
           [Tv (VarT T)])
       (cond
         [(equal? Tv t3) (succeed)]
         [(occurs? Tv t3) (type-error expr Tv t3)]
         [else (set-box! T (some t3))]))]))
```

Type Unification

```
(define (unify-assert! tau type-val expr)
  (unless (equal? tau type-val)
    (type-error expr tau type-val)))
```

Type Unification Helpers

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

Type Unification Helpers

```
(define (occurs? r t)
  (type-case Type t
    [(NumT) #f]
    [(BoolT) #f]
    [(ArrowT a b)
     (or (occurs? r a)
         (occurs? r b))]
    [(VarT is)
     (or (equal? r t)
         (type-case (Optionof Type) (unbox is)
           [(none) #f]
           [(some t2) (occurs? r t2)]))]))
```

TIFAE Type Checker

```
(define (typecheck [fae : FAE] [env : TypeEnv]) : Type
  (type-case FAE fae
      ...
      [(Num n) (NumT)]
      ...))
```

TIFAE Type Checker

```
(define (typecheck [fae : FAE] [env : TypeEnv]) : Type
  (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 : FAE] [env : TypeEnv]) : Type
  (type-case FAE fae
     ...
     [(Id name) (type-lookup name env)]
     [(Fun name te body)
      (let* ([arg-type (parse-type te)]
             [res-type (typecheck
                         body
                         (aBind name arg-type env))])
        (ArrowT arg-type res-type))]
     ...))
```

TIFAE Type Checker

```
(define (typecheck [fae : FAE] [env : TypeEnv]) : Type
  (type-case FAE fae
    ...
    [(Call fn arg)
     (let ([r-type (VarT (box (none)))]
           [a-type (typecheck arg env)]
           [fn-type (typecheck fn env)])
       (begin
         (unify! (ArrowT a-type r-type) fn-type fn)
         r-type))]
    ...))
```

TIFAE Type Checker

```
(define (typecheck [fae : FAE] [env : TypeEnv]) : Type
  (type-case FAE fae
    ...
    [(If0 test-expr then-expr else-expr)
    (let ([test-ty (typecheck test-expr env)]
          [then-ty (typecheck then-expr env)]
          [else-ty (typecheck else-expr env)])
      (begin
        (unify! test-ty (NumT) test-expr)
        (unify! then-ty else-ty else-expr)
        then-ty))]
    ...))
```

TIFAE Type Checker

```
(define (typecheck [fae : FAE] [env : TypeEnv]) : Type
  (type-case FAE fae
     ...
     [(Rec name ty rhs-expr body-expr)
      (let* ([type-ann (parse-type ty)]
             [rhs-typ (typecheck rhs-expr new-ds)]
             [new-ds (aBind name rhs-ty env)])
        (begin
          (unify! type-ann rhs-ty rhs-expr)
          (typecheck body-expr new-ds)))]
     ...))
```