Types and evaluation

- Why is a type system useful?

Types and evaluation

- Why is a type system useful?
- What information can a type system give us?

Types and evaluation

- Why is a type system useful?
- What information can a type system give us?
- What is the relationship between:

$$\Gamma \vdash \mathbf{e} : \tau$$

and

```
; interp-expr e -> v
;
```

Types and evaluation

- Why is a type system useful?
- What information can a type system give us?
- What is the relationship between:

$$\Gamma \vdash \mathbf{e} : \tau$$

and

```
; interp-expr e -> v
;
```

- We'd like types to tell us something useful about the behavior of our runtime system.

Type Soundness

If

$\varnothing \vdash$ **e** : τ

then

(interp-expr e) = v and

if τ = **num** then **v** is a num

if τ = (τ_1 -> τ_2) then **v** is **'procedure**

Type Soundness

If

$\varnothing \vdash$ **e** : τ

then

(interp-expr e) = v and

if τ = **num** then **v** is a num

if τ = $(\tau_1$ -> $\tau_2)$ then **v** is **'procedure**

- Is this true for TFAE?

Type Soundness

If

$\varnothing \vdash$ **e** : τ

then

(interp-expr e) = v and

if τ = **num** then **v** is a num

if τ = (τ_1 -> τ_2) then **v** is **'procedure**

- Is this true for TFAE?
- What about other languages?

Type Soundness

If we *only* allow programs such that

```
(interp-expr e) = v
```

Type Soundness

If we *only* allow programs such that

```
(interp-expr e) = v
```

```
{+ 5 false}
```

We'd like to rule out things like this, and we do.

Type Soundness

If we *only* allow programs such that

```
(interp-expr e) = v
```

```
{/ 5 ...}
```

We'd probably like to allow this.

But what if `...` evaluates to 0?

Type Soundness

If we *only* allow programs such that

```
(interp-expr e) = v
```

```
{/ 5 ...}
```

We'd probably like to allow this.

But what if `...` evaluates to 0?

We're also forced rule out programs that don't terminate, or may not terminate.

Type Soundness

If

$\varnothing \vdash$ **e** : τ and

(interp-expr e) = v

then

if τ = **num** then **v** is a num

if τ = $(\tau_1$ -> $\tau_2)$ then **v** is **'procedure**

Type Soundness

If

$\varnothing \vdash$ **e** : τ and

(interp-expr e) = v

then

if τ = **num** then **v** is a num

if τ = $(\tau_1$ -> $\tau_2)$ then **v** is **'procedure**

- We made the condition on interpreted values a premise

Type Soundness

If

$\varnothing \vdash$ **e** : τ and

(interp-expr e) = v

then

if τ = **num** then **v** is a num

if τ = $(\tau_1$ -> $\tau_2)$ then **v** is **'procedure**

- We made the condition on interpreted values a premise
- This allows the programs we want to allow, but considerably weakens the statement of type soundness

Recursion

```
{with {mk-rec {fun {body}
                   {{fun {fX} {fX fX}}
                    {fun {fX}
                         {{fun {f} {body f}}
                          {fun {x} {{fX fX} x}}}}}}}
      {with {fib {mk-rec
                  {fun {fib}
                       {fun {n}
                            {if0 n
                                 1
                                 {if0 {- n 1}
                                      1
                                      {+ {fib {- n 1}}
                                         {fib {- n 2}}}}}}}}}
            {fib 4}}}
```

Typed Recursion

```
{with {mk-rec : (((num -> num) -> (num -> num)) -> (num -> num))
              {fun {body : ((num -> num) -> (num -> num))}
                   {{fun {fX : … -> (num -> num)} {fX fX}}
                    {fun {fX : … -> (num -> num)}
                         {{fun {f : (num -> num)} {body f}}
                          {fun {x : num} {{fX fX} x}}}}}}}
      {with {fib : (num -> num)
                   {mk-rec
                    {fun {fib : (num -> num)}
                         {fun {n : num}
                              {if0 n
                                   1
                                   {if0 {- n 1}
                                        1
                                        {+ {fib {- n 1}}
                                           {fib {- n 2}}}}}}}}}
            {fib 4}}}
```

Typed Recursion

```
{with {mk-rec : (((num -> num) -> (num -> num)) -> (num -> num))
              {fun {body : ((num -> num) -> (num -> num))}
                   {{fun {fX : … -> (num -> num)} {fX fX}}
                    {fun {fX : … -> (num -> num)}
                         {{fun {f : (num -> num)} {body f}}
                          {fun {x : num} {{fX fX} x}}}}}}}
      {with {fib : (num -> num)
                   {mk-rec
                    {fun {fib : (num -> num)}
                         {fun {n : num}
                              {if0 n
                                   1
                                   {if0 {- n 1}
                                        1
                                        {+ {fib {- n 1}}
                                           {fib {- n 2}}}}}}}}}
            {fib 4}}}
```

Nothing works in place of the …

Extending the Type System

When the type system rejects your prefectly good program, it may be time to extend the type system

In this case, we can add **`rec`** as a core form, again

```
{rec {fib : (num -> num)
           {fun {n : num}
                {if0 n
                     1
                     {if0 {- n 1}
                          1
                          {+ {fib {- n 1}}
                             {fib {- n 2}}}}}}}}
  {fib 4}}
```

TRCFAE Grammar

```
<TRCFAE>  ::=  <num>
            |  {+ <TRCFAE> <TRCFAE>}
            |  {- <TRCFAE> <TRCFAE>}
            |  <id>
            |  {fun {<id> : <TE>} <TRCFAE>}
            |  {<TRCFAE> <TRCFAE>}
            |  {if0 <TRCFAE> <TRCFAE> <TRCFAE>}          NEW
            |  {rec {<id> : <TE> <TRCFAE>} <TRCFAE>}     NEW
<TE>      ::=  num
            |  (<TE> -> <TE>)
```

TRCFAE Datatypes

```
(define-type TFAE
  ...
  [if0 (test-expr : TFAE)
       (then-expr : TFAE)
       (else-expr : TFAE)]
  [rec (name : symbol)
    (ty : Type)
    (rhs-expr : TFAE)
    (body-expr : TFAE)])
```

TRCFAE Interpreter

```
(define (interp a-fae ds)
  (type-case TFAE a-fae
    ...
    [if0 (test-expr then-expr else-expr)
         (if (numzero? (interp test-expr ds))
             (interp then-expr ds)
             (interp else-expr ds))]
    [rec (bound-id type named-expr body-expr)
      (local [(define value-holder (box (numV 42)))
              (define new-ds (aRecSub bound-id
                                      value-holder
                                      ds))]
        (begin
          (set-box! value-holder (interp named-expr new-ds))
          (interp body-expr new-ds)))]))
```

TRCFAE Interpreter Lookup

```
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "free variable")]
    [aSub (sub-name val rest-ds)
          (if (symbol=? sub-name name)
              val
              (lookup name rest-ds))]
    [aRecSub (sub-name val-box rest-ds)
             (if (symbol=? sub-name name)
                 (unbox val-box)
                 (lookup name rest-ds))]))
```

TRCFAE Grammar

```
<TRCFAE>  ::=  <num>
            |  {+ <TRCFAE> <TRCFAE>}
            |  {- <TRCFAE> <TRCFAE>}
            |  <id>
            |  {fun {<id> : <TE>} <TRCFAE>}
            |  {<TRCFAE> <TRCFAE>}
            |  {if0 <TRCFAE> <TRCFAE> <TRCFAE>}        NEW
            |  {rec {<id> : <TE> <TRCFAE>} <TRCFAE>}   NEW
<TE>      ::=  num
            |  (<TE> -> <TE>)
```

TRCFAE Grammar

```
<TRCFAE>  ::=  <num>
            |  {+ <TRCFAE> <TRCFAE>}
            |  {- <TRCFAE> <TRCFAE>}
            |  <id>
            |  {fun {<id> : <TE>} <TRCFAE>}
            |  {<TRCFAE> <TRCFAE>}
            |  {if0 <TRCFAE> <TRCFAE> <TRCFAE>}        NEW
            |  {rec {<id> : <TE> <TRCFAE>} <TRCFAE>}   NEW
<TE>      ::=  num
            |  (<TE> -> <TE>)
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \mathit{num} \qquad \Gamma \vdash \mathbf{e}_2 : \tau_0 \qquad \Gamma \vdash \mathbf{e}_3 : \tau_0}{\Gamma \vdash \{\mathtt{if0}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3\} : \tau_0}$$

TRCFAE Grammar

```
<TRCFAE>  ::=  <num>
            |  {+ <TRCFAE> <TRCFAE>}
            |  {- <TRCFAE> <TRCFAE>}
            |  <id>
            |  {fun {<id> : <TE>} <TRCFAE>}
            |  {<TRCFAE> <TRCFAE>}
            |  {if0 <TRCFAE> <TRCFAE> <TRCFAE>}        NEW
            |  {rec {<id> : <TE> <TRCFAE>} <TRCFAE>}   NEW
<TE>      ::=  num
            |  (<TE> -> <TE>)
```

$$\frac{\Gamma[\texttt{<id>}\leftarrow\tau_0] \vdash \mathbf{e}_0 : \tau_0 \qquad \Gamma[\texttt{<id>}\leftarrow\tau_0] \vdash \mathbf{e}_1 : \tau_1}{\Gamma \vdash \{\texttt{rec } \{\texttt{<id>} : \tau_0\ \mathbf{e}_0\}\ \mathbf{e}_1\} : \tau_1}$$

TRCFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TFAE fae
      ...
      [if0 (test-expr then-expr else-expr)
           (type-case Type (typecheck test-expr env)
             [numT () (local [(define test-ty
                                (typecheck then-expr env))]
                        (if (equal? test-ty
                                    (typecheck else-expr env))
                            test-ty
                            (type-error else-expr
                                        (to-string test-ty))))]
             [else (type-error test-expr "num")])])))
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \mathit{num} \qquad \Gamma \vdash \mathbf{e}_2 : \tau_0 \qquad \Gamma \vdash \mathbf{e}_3 : \tau_0}{\Gamma \vdash \{\mathtt{if0}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3\} : \tau_0}$$

TRCFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TFAE fae
      ...
      [rec (name ty rhs-expr body-expr)
        (local [(define rhs-ty (parse-type ty))
                (define new-ds (aBind name
                                      rhs-ty
                                      env))]
          (if (equal? rhs-ty (typecheck rhs-expr new-ds))
              (typecheck body-expr new-ds)
              (type-error rhs-expr (to-string rhs-ty))))])))
```

$$\frac{\Gamma[\texttt{<id>}\leftarrow\tau_0] \vdash \texttt{e}_0 : \tau_0 \qquad \Gamma[\texttt{<id>}\leftarrow\tau_0] \vdash \texttt{e}_1 : \tau_1}{\Gamma \vdash \{\texttt{rec } \{\texttt{<id>} : \tau_0 \ \texttt{e}_0\} \ \texttt{e}_1\} : \tau_1}$$

Sum Types

```
(define-type NorfSum
  [left (n : number)]
  [right (f : (number -> number))])
(let ([norf (left 5)])
  (type-case NorfSum norf
    [left (x)
          (+ x 1)]
    [right (f)
           (f 0)]))
```

- Typed PLAI's **`define-type`** constructs a variant type.
- A value of this type can hold many different types, which are differentiated by labels.

Sum Types

```
(define-type NorfSum
  [left (n : number)]
  [right (f : (number -> number))])
(let ([norf (left 5)])
  (type-case NorfSum norf
    [left (x)
          (+ x 1)]
    [right (f)
           (f 0)]))
```

- A sum type is like a variant type, except that there are only two variants, and the tags are always `left` and `right`.

TSFAE

```
{with {norf {left 5 as (num + (num -> num))}}
      {+ 1
         {case norf
           [left x
                 {+ x 1}]
           [right f
                  {f 0}]}}}
```

TSFAE Grammar

```
<TSFAE>  ::=  <num>
           |  {+ <TSFAE> <TSFAE>}
           |  {- <TSFAE> <TSFAE>}
           |  <id>
           |  {fun {<id> : <TE>} <TPFAE>}
           |  {<TSFAE> <TSFAE>}
           |  {left <TSFAE> as <TE>}          NEW
           |  {right <TSFAE> as <TE>}         NEW
           |  {case <TSFAE>                   NEW
                [left <id> <TSFAE>]
                [right <id> <TSFAE>]}
<TE>     ::=  num
           |  (<TE> -> <TE>)
           |  (<TE> + <TE>)                   NEW
```

TSFAE Grammar

```
<TPFAE>  ::=  ...
           |  {left <TSFAE> as <TE>}        NEW
           |  {right <TSFAE> as <TE>}       NEW
           |  {case <TSFAE>                  NEW
                [left <id> <TSFAE>]
                [right <id> <TSFAE>]}
<TE>     ::=  num
           |  (<TE> -> <TE>)
           |  (<TE> + <TE>)                 NEW
```

TSFAE Grammar

```
<TPFAE>  ::=  ...
          |   {left <TSFAE> as <TE>}        NEW
          |   {right <TSFAE> as <TE>}       NEW
          |   {case <TSFAE>                  NEW
                [left <id> <TSFAE>]
                [right <id> <TSFAE>]}
<TE>     ::=  num
          |   (<TE> -> <TE>)
          |   (<TE> + <TE>)                  NEW
```

$$\frac{\Gamma \vdash \mathbf{e} : \tau_1}{\Gamma \vdash \{\mathbf{left}\ \mathbf{e}\ \mathbf{as}\ (\tau_1 + \tau_2)\} : (\tau_1 + \tau_2)}$$

TSFAE Grammar

```
<TPFAE>  ::=  ...
          |   {left <TSFAE> as <TE>}        NEW
          |   {right <TSFAE> as <TE>}       NEW
          |   {case <TSFAE>                  NEW
                [left <id> <TSFAE>]
                [right <id> <TSFAE>]}
<TE>     ::=  num
          |   (<TE> -> <TE>)
          |   (<TE> + <TE>)                  NEW
```

$$\frac{\Gamma \vdash \texttt{e} : \tau_2}{\Gamma \vdash \texttt{\{right e as } (\tau_1 \texttt{ + } \tau_2)\texttt{\}} : (\tau_1 \texttt{ + } \tau_2)}$$

TSFAE Grammar

```
<TPFAE>  ::=  ...
           |  {left <TSFAE> as <TE>}        NEW
           |  {right <TSFAE> as <TE>}       NEW
           |  {case <TSFAE>                  NEW
                [left <id> <TSFAE>]
                [right <id> <TSFAE>]}
<TE>     ::=  num
           |  (<TE> -> <TE>)
           |  (<TE> + <TE>)                 NEW
```

$$\frac{\Gamma \vdash \mathbf{e} : (\tau_1 + \tau_2) \qquad \Gamma\ [\langle \mathbf{id} \rangle_1 \leftarrow \tau_1] \vdash \mathbf{e}_1 : \tau \qquad \Gamma\ [\langle \mathbf{id} \rangle_2 \leftarrow \tau_2] \vdash \mathbf{e}_2 : \tau}{\Gamma \vdash \{\mathbf{case}\ \mathbf{e}\ [\mathbf{left}\ \langle \mathbf{id} \rangle_1\ \mathbf{e}_1]\ [\mathbf{right}\ \langle \mathbf{id} \rangle_2\ \mathbf{e}_2]\} : \tau}$$

TSFAE Datatypes

```
(define-type TFAE
  ...
  [lft (e : TFAE)
       (typ : Type)]
  [rgt (e : TFAE)
       (typ : Type)]
  [cse (e : TFAE)
       (l-id : symbol)
       (l-e : TFAE)
       (r-id : symbol)
       (r-e : TFAE)])
```

TSFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TFAE fae
      ...
      [lft (exp typ)
           (type-case Type typ
             [sumT (l r)
                   (if (equal? (type-check exp env) l)
                       typ
                       (type-error test-expr "lft"))]
             [else
              (type-error test-expr "lft")])])))
```

$$\frac{\Gamma \vdash \mathbf{e} : \tau_1}{\Gamma \vdash \{\mathbf{left}\ \mathbf{e}\ \mathbf{as}\ (\tau_1 + \tau_2)\} : (\tau_1 + \tau_2)}$$

TSFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TFAE fae
      ...
      [cse (e l-id l-e r-id r-e)
           (local [(define e-typ (type-check e env))]
             (type-case Type e-typ
               [sumT (l r)
                     ...]
               [else
                (type-error test-expr "cse")]))])))
```

$$\frac{\Gamma \vdash \mathbf{e} : (\tau_1 + \tau_2) \qquad \Gamma\ [\langle \mathbf{id} \rangle_1 \leftarrow \tau_1] \vdash \mathbf{e}_1 : \tau \qquad \Gamma\ [\langle \mathbf{id} \rangle_2 \leftarrow \tau_2] \vdash \mathbf{e}_2 : \tau}{\Gamma \vdash \{\mathbf{case}\ \mathbf{e}\ [\mathbf{left}\ \langle \mathbf{id} \rangle_1\ \mathbf{e}_1]\ [\mathbf{right}\ \langle \mathbf{id} \rangle_2\ \mathbf{e}_2]\} : \tau}$$

TSFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TFAE fae
      ...
      [cse (e l-id l-e r-id r-e)
           (local [(define e-typ (type-check e env))]
             (type-case Type e-typ
               [sumT (l r)
                     (local [(define l-typ
                               (type-check l-e (aEnv l-id l env)))
                             (define r-typ
                               (type-check r-e (aEnv r-id r env)))]
                       (if (equal? l-typ r-typ)
                           l-typ
                           (type-error test-expr "cse")))]
               [else
                (type-error test-expr "cse")]))])))
```