Type System Extensions

When Type Systems Come Up Short

- Type systems are inherently conservative
  - Not sure whether a program is ok → have to reject
- So it's possible (and sadly common) to write a program which is ok according to the grammar
  - and which would run just fine in an interpreter
  - but which the type system can't prove is ok
    - and so rejects

When Type Systems Come Up Short

- To get around that, we can extend the language (and the type system) to express these programs differently
- Additional language construct
  → additional typing rule
  → additional way the type checker can prove things!
- Even if, strictly speaking, we wouldn't need these constructs just for the interpreter

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 : (((number -> number) -> (number -> number))
                 -> (number -> number))
                {fun {body : ((number -> number)
                              -> (number -> number))}
                     {{fun {fX : … -> (number -> number)} {fX fX}}
                      {fun {fX : … -> (number -> number)}
                           {{fun {f : (number -> number)} {body f}}
                            {fun {x : number} {{fX fX} x}}}}}}}
      {with {fib : (number -> number)
                   {mk-rec
                    {fun {fib : (number -> number)}
                         {fun {n : number}
                              {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 : (number -> number)
           {fun {n : number}
                {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>      ::=  number
            |  (<TE> -> <TE>)
```

TRCFAE Datatypes

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

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>      ::=  number
            |  (<TE> -> <TE>)
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \textit{number} \qquad \Gamma \vdash \mathbf{e}_2 : \tau_0 \qquad \Gamma \vdash \mathbf{e}_3 : \tau_0}{\Gamma \vdash \{\texttt{if0}\ \mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3\} : \tau_0}$$

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>      ::=  number
            |  (<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)
             [numberT () (local [(define then-type
                                   (typecheck then-expr env))]
                           (if (equal? then-type
                                       (typecheck else-expr env))
                               then-type
                               (error 'typecheck "type mismatch")))]
             [else (error 'typecheck "expected number")])])))
```

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

TRCFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TFAE fae
      ...
      [rec (name ty named-expr body)
           (local [(define new-env (aBind name ty env))]
             (if (equal? ty (typecheck named-expr new-env))
                 (typecheck body new-env)
                 (error 'typecheck "type mismatch")))])))
```

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

Sums of Products

- Sometimes we add constructs to our language just because we want to be able to do more.
- In these cases also, we need to add typing rules for the new constructs.
- One thing we may want to be able to do: **`define-type`**

```
(define-type AE
  [num (n : number)]
  [add (l : AE) (r : AE)]
  [sub (l : AE) (r : AE)])
```

Sums of Products

- **`define-type`** defines "sums of products"

```
(define-type AE
  [num (n : number)]
  [add (l : AE) (r : AE)]
  [sub (l : AE) (r : AE)])
```

- There are multiple alternatives: **`num`**, **`add`**, **`sub`** (algebraic sum, think set union)
- Each one may have multiple fields: e.g., **`add`** has both **`l`** and **`r`** (algebraic product, think cartesian product)

- **HW9:** a specific kind of sum-of-products, cons lists
- **Now:** just the sum part, but kept general

Sum Types

```
(define-type NorFSum
  [left (n : number)]
  [right (f : (number -> number))])

(let ([s (left 5)])
  (+ 1
     (type-case NorFSum s
       [left  (x) (+ x 1)]
       [right (f) (f 0)])))
```

To keep things simple:

- Only two alternatives
- Always named `left` and `right`
- Single field (no products)

TSFAE

```
(let ([s (left 5)])
  (+ 1
     (type-case NorFSum s
       [left  (x) (+ x 1)]
       [right (f) (f 0)])))
```

No **`define-type`**, instead constructors specify the type.

`case` is like a single-field **`type-case`** without the type.

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

TSFAE Grammar

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

TSFAE Grammar

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

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

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

TSFAE Grammar

```
<TSFAE>  ::=  ...
           |  {left <TSFAE> as <TE>}                    NEW
           |  {right <TSFAE> as <TE>}                   NEW
           |  {case <TSFAE>                             NEW
                [left  (<id>) <TSFAE>]
                [right (<id>) <TSFAE>]}
<TE>     ::=  number
           |  (<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)
       (ty : Type)]
  [rgt (e : TFAE)
       (ty : Type)]
  [cse (e : TFAE)
       (l-name : symbol)
       (l-body : TFAE)
       (r-name : symbol)
       (r-body : TFAE)])
```

TSFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TFAE fae
      ...
      [lft (e ty)
           (type-case Type ty
             [sumT (l r)
                   (if (equal? (type-check e env) l)
                       ty
                       (error 'typecheck "type mismatch"))]
             [else
              (error 'typecheck "type mismatch")])])))
```

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

TSFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TFAE fae
      ...
      [cse (e l-name l-body r-name r-body)
           (local [(define e-ty (typecheck e env))]
             (type-case Type e-ty
               [sumT (l r)
                     ...]
               [else
                (error 'typecheck "expected sum type")]))])))
```

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

TSFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case TFAE fae
      ...
      [cse (e l-name l-body r-name r-body)
           (local [(define e-ty (typecheck e env))]
             (type-case Type e-ty
               [sumT (l r)
                     (local [(define l-ty
                               (typecheck l-body
                                          (aBind l-name l env)))
                             (define r-ty
                               (typecheck r-body
                                          (aBind r-name r env)))]
                       (if (equal? l-ty r-ty)
                           l-ty
                           (error 'typecheck "type mismatch")))]
               [else
                (error 'typecheck "expected sum type")]))])))
```