From Lecture 15: FAE with pairs

```
<TPFAE>  ::=  <num>
          |   ...
          |   {fun {<id> : <TE>} <TPFAE>}
          |   {<TPFAE> <TPFAE>}
          |   {pair <TPFAE> <TPFAE>}        NEW
          |   {fst <TPFAE>}                 NEW
          |   {snd <TPFAE>}                 NEW

<TE>     ::=  num
          |   bool
          |   (<TE> -> <TE>)
          |   (<TE> * <TE>)                 NEW
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \tau_1 \qquad \Gamma \vdash \mathbf{e}_2 : \tau_2}{\Gamma \vdash \{\mathbf{pair}\ \mathbf{e}_1\ \mathbf{e}_2\} : (\tau_1 \times \tau_2)}$$

From Lecture 15: FAE with pairs

```
<TPFAE>  ::=  <num>
          |   ...
          |   {fun {<id> : <TE>} <TPFAE>}
          |   {<TPFAE> <TPFAE>}
          |   {pair <TPFAE> <TPFAE>}        NEW
          |   {fst <TPFAE>}                 NEW
          |   {snd <TPFAE>}                 NEW

<TE>     ::=  num
          |   bool
          |   (<TE> -> <TE>)
          |   (<TE> * <TE>)                 NEW
```

$$\frac{\Gamma \vdash \texttt{e} : (\tau_1 \times \tau_2)}{\Gamma \vdash \texttt{\{fst e\}} : \tau_1}$$

From Lecture 15: FAE with pairs

```
<TPFAE>  ::=  <num>
          |   ...
          |   {fun {<id> : <TE>} <TPFAE>}
          |   {<TPFAE> <TPFAE>}
          |   {pair <TPFAE> <TPFAE>}        NEW
          |   {fst <TPFAE>}                 NEW
          |   {snd <TPFAE>}                 NEW

<TE>     ::=  num
          |   bool
          |   (<TE> -> <TE>)
          |   (<TE> * <TE>)                 NEW
```

$$\frac{\Gamma \vdash \texttt{e} : (\tau_1 \times \tau_2)}{\Gamma \vdash \texttt{\{snd e\}} : \tau_2}$$

Recursive Datatypes

```
{with {empty : (num * …)
             {pair 0 ...}}
  {with {cons : (num -> ((num * …) -> (num * …)))
             {fun {x : num}
                  {fun {r : (num * …)}
                       {pair 1 {pair x r}}}}}
    {{cons 1} {{cons 2} {{cons 3} empty}}}}}
```

Stuck again with …

Recursive Datatypes

Add **with-type** and **type-case**:

```
{with-type {numlist {empty num}
                    {cons (num * numlist)}}
  {rec {len : (numlist -> num)
            {fun {l : numlist}
                 {type-case numlist l
                   {{empty n} 0}
                   {{cons fxr} {+ 1 {len {snd fxr}}}}}}}
       {len {cons {pair 1 {cons {pair 2 {empty 0}}}}}}}}
```

TVRCFAE Grammar

```
<TVRCFAE>  ::=  <num>
            |   ...
            |   {rec {<id> : <TE> <TVRCFAE>} <TVRCFAE>}
            |   {with-type {<tyid> {<id>  <TE>}                    NEW
                                   {<id>  <TE>}}
                      <TVRCFAE>}
            |   {type-case <tyid> <TVRCFAE>                        NEW
                  {<id> {<id>} <TVRCFAE>}
                  {<id> {<id>} <TVRCFAE>}}

<TE>       ::=  num
            |   (<TE> -> <TE>)
            |   <tyid>                                             NEW
```

Well-Formed Type Expressions

- Might be ok:

```
{with-type {fruit {apple num}
                  {banana (num -> num)}}
           ... {fun {x : fruit} ...} ...}
```

- Not ok:

```
{fun {x : fruit} ...}
```

Well-Formed Type Expressions (type-id lookup)

$$\Gamma \vdash \texttt{num} \qquad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash (\tau_1 \texttt{ -> } \tau_2)}$$

$$[\ \ldots\ \texttt{<tyid>} = \texttt{<id>}_1@\tau_1 + \texttt{<id>}_2@\tau_2\ \ \ldots\ ] \vdash \texttt{<tyid>}$$

TVRCFAE Type Checker

$$\frac{\Gamma' = \Gamma[\ \texttt{<tyid>} = \texttt{<id>}_1@\tau_1 + \texttt{<id>}_2@\tau_2,\ \texttt{<id>}_1 \leftarrow (\tau_1 \rightarrow \texttt{<tyid>}),\ \texttt{<id>}_2 \leftarrow (\tau_2 \rightarrow \texttt{<tyid>})\ ] \qquad \Gamma' \vdash \tau_1 \qquad \Gamma' \vdash \tau_2 \qquad \Gamma' \vdash \texttt{e} : \tau_0}{\Gamma \vdash \texttt{\{with-type \{<tyid> \{<id>}_1\ \ \tau_1\texttt{\} \{<id>}_2\ \ \tau_2\texttt{\}\} e\}} : \tau_0}$$

$$\frac{\Gamma' = \Gamma[\ \texttt{<tyid>} = \texttt{<id>}_1@\tau_1 + \texttt{<id>}_2@\tau_2\ ] \qquad \Gamma' \vdash \texttt{e}_0 : \texttt{<tyid>} \qquad \Gamma'[\ \texttt{<id>}_3 \leftarrow \tau_1\ ] \vdash \texttt{e}_1 : \tau_0 \qquad \Gamma'[\ \texttt{<id>}_4 \leftarrow \tau_2\ ] \vdash \texttt{e}_2 : \tau_0}{\Gamma' \vdash \texttt{\{type-case <tyid> e}_0\ \texttt{\{<id>}_1\ \texttt{\{<id>}_3\texttt{\} e}_1\texttt{\} \{<id>}_2\ \texttt{\{<id>}_4\texttt{\} e}_2\texttt{\}\}} : \tau_0}$$

Warning: later, we'll discuss why the `with-type` rule is not quite right

TVRCFAE Expression Datatypes

```
(define-type FAE
  [WithType (name : Symbol)
            (var1-name : Symbol)
            (var1-ty : TE)
            (var2-name : Symbol)
            (var2-ty : TE)
            (body-expr : FAE)]
  [TypeCase (name : Symbol)
            (dispatch-expr : FAE)
            (var1-name : Symbol)
            (bind1-name : Symbol)
            (rhs1-expr : FAE)
            (var2-name : Symbol)
            (bind2-name : Symbol)
            (rhs2-expr : FAE)])

(define-type TE
  ...
  [IdTE (name : symbol)])
```

TVRCFAE Value and Environment Datatypes

```
(define-type FAE-Value
  ...
  [VariantV (right? : Boolean)
            (val : FAE-Value)]
  [ConstructorV (right? : Boolean)])

(define-type TypeEnv
  ...
  [tBind (name : symbol)
         (var1-name : symbol)
         (var1-type : Type)
         (var2-name : symbol)
         (var2-type : Type)
         (rest : TypeEnv)])
```

TVRCFAE Interpreter

```
(define (eval a-fae ds)
  (type-case FAE a-fae
    ...
    [(WithType type-name
               var1-name var1-te
               var2-name var2-te body-expr)
     (eval body-expr (aSub var1-name (ConstructorV #f)
                           (aSub var2-name (ConstructorV #t) env)))]
    ...))
```

TVRCFAE Interpreter

```
(define (eval a-fae ds)
  (type-case FAE a-fae
    ...
  [(Call fun-expr arg-expr)
   (let* ([fun-val (eval fun-expr env)]
          [arg-val (eval arg-expr env)])
     (type-case FAE-Value fun-val
       [(ClosureV param body env)
       (eval body (aSub param arg-val env))]
       [(ConstructorV right?)
        (VariantV right? arg-val)]
       [else (error 'eval "not callable")]))]
    ...))
```

TVRCFAE Interpreter

```
(define (eval a-fae ds)
  (type-case FAE a-fae
    ...
    [(TypeCase ty dispatch-expr
               var1-name var1-id var1-rhs
               var2-name var2-id var2-rhs)
     (type-case FAE-Value (eval dispatch-expr env)
       [(VariantV right? val)
        (if (not right?)
            (eval var1-rhs (aSub var1-id val env))
            (eval var2-rhs (aSub var2-id val env)))]
       [else (error 'eval "not a variant result")])]
    ...))
```

TVRCFAE Type Lookup

```
(define (type-lookup name-to-find env)
  (type-case TypeEnv env
    [(mtEnv) (error 'type-lookup "free variable, so no type")]
    [(aBind name ty rest)
     (if (equal? name-to-find name)
         ty
         (type-lookup name-to-find rest))]
    [(tBind name var1-name var1-ty var2-name var2-ty rest)
     (type-lookup name-to-find rest)]))
```

TVRCFAE Type Lookup

```
(define (find-type-id name-to-find env)
  (type-case TypeEnv env
    [(mtEnv) (error 'get-type "free type name, so no type")]
    [(aBind name ty rest)
     (find-type-id name-to-find rest)]
    [(tBind name var1-name var1-ty var2-name var2-ty rest)
     (if (equal? name-to-find name)
         env
         (find-type-id name-to-find rest))]))
```

TVRCFAE Type-Expression Checking

```
(define (validtype ty env)
  (type-case Type ty
    [(NumT) (mtEnv)]
    [(BoolT) (mtEnv)]
    [(ArrowT a b)
     (begin
       (validtype a env)
       (validtype b env))]
    [(IdT id) (find-type-id id env)]))
```

TVRCFAE Type Checking

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

TVRCFAE Type Checking

```
(define (typecheck [fae : FAE] [env : TypeEnv]) : Type
  (type-case FAE fae
    ...
    [(WithType type-name var1-name var1-te var2-name var2-te body-expr)
     (let* ([var1-ty (parse-type var1-te)]
            [var2-ty (parse-type var2-te)]
            [new-env (tBind type-name var1-name var1-ty
                            var2-name var2-ty env)])
       (begin
         (validtype var1-ty new-env)
         (validtype var2-ty new-env)
         (typecheck body-expr
                    (aBind var1-name
                           (ArrowT var1-ty (IdT type-name))
                           (aBind var2-name
                                  (ArrowT var2-ty (IdT type-name))
                                  new-env)))))]
    ...))
```

TVRCFAE Type Checking

```
(define (typecheck [fae : FAE] [env : TypeEnv]) : Type
  (type-case FAE fae
      ...
      [(TypeCase type-name dispatch-expr
                 var1-name var1-id var1-rhs var2-name var2-id var2-rh
       (let ([type-binding (find-type-id type-name env)]
             [expr-type (typecheck dispatch-expr env)])
         (type-case Type expr-type
           [(IdT name)
            (begin
              (unless (equal? name type-name)
                (type-error dispatch-expr (to-string type-name)))
              (matching-variant-names type-binding var1-name var2-nam
              (matching-variant-types type-binding var1-id var1-rhs
                                      var2-id var2-rhs env))]
           [else (type-error dispatch-expr (to-string type-name))]))]
```

TVRCFAE Type Checking

```
(define (matching-variant-types type-binding
                                var1-id var1-rhs var2-id var2-rhs env
  (let* ([var1-ty (tBind-var1-type type-binding)]
         [var2-ty (tBind-var2-type type-binding)]
         [env1 (aBind var1-id var1-ty env)]
         [rhs1-ty (typecheck var1-rhs env1)]
         [env2 (aBind var2-id var2-ty env)])
    (type-assert (list var2-rhs) rhs1-ty env2 rhs1-ty)))
```

TVRCFAE Type Checking

```
(define (matching-variant-names type-binding var1-name var2-name fae)
  (unless
      (and (equal? var1-name (tBind-var1-name type-binding))
           (equal? var2-name (tBind-var2-name type-binding)))
    (type-error fae "matching variant names")))
```

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, then divide-by-zero errors may be ok:

If $\varnothing \vdash \mathbf{e} : \tau$, 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 TVRCFAE

TCRCFAE has a bug, too:

```
{call {with-type {foo {a num} {b num}}
        {fun {x : foo}
             {type-case foo x
               {{a n} n}
               {{b n} n}}}}
      {with-type {foo {c (num -> num)} {d num}}
        {c {fun {y : num} y}}}}
```

Solution 1: no local type declarations

Type Soundness in TVRCFAE

TCRCFAE has a bug, too:

```
{call {with-type {foo {a num} {b num}}
        {fun {x : foo}
             {type-case foo x
               {{a n} n}
               {{b n} n}}}}
      {with-type {foo {c (num -> num)} {d num}}
        {c {fun {y : num} y}}}}
```

Solution 2: don't let **<tyid>** escape **with-type**

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