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}}}
```

Nothing works in place of ...

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}}
```

We'll add **if0**, too, while we're at it

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 FAE
  ...
  [if0 (test-expr : FAE)
       (then-expr : FAE)
       (else-expr : FAE)]
  [rec (name : symbol)
       (ty : TE)
       (rhs-expr : FAE)
       (body-expr : FAE)])
```

TRCFAE Interpreter

```
(define (interp a-fae ds)
  (type-case FAE 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 Type Checker

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE 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")])])))
```

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

TRCFAE 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))]
             (if (equal? rhs-ty (typecheck rhs-expr new-ds))
                 (typecheck body-expr new-ds)
                 (type-error rhs-expr (to-string rhs-ty))))])))
```

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

Variants

```
{with {left : (num -> (num * num))
            {fun {x : num}
                 {pair 0 x}}}
  {with {right : (num -> (num * num))
               {fun {x : num}
                    {pair 1 x}}}
    {with {displacement : ((num * num) -> num)
                        {fun {p : (num * num)}
                             {if0 {fst p}
                                  {- 0 {snd p}}
                                  {snd p}}}}
      {displacement {left 5}}}}}
```

Variants

```
{with {grade : (num -> (num * (num * bool)))
             {fun {x : num}
                  {pair 0 {pair x false}}}}
  {with {pf : (num -> (num * (num * bool)))
            {fun {y : bool}
                 {pair 1 {pair 0 y}}}}
    {with {pass? : ((num * (num * bool)) -> bool)
                 {fun {p : (num * (num * bool))}
                      {if0 {fst p}
                           {> {fst {snd p}} 70}
                           {snd {snd p}}}}}}
      {pass? {grade 96}}}}}
```

Have to make up a value for the other type, but this can be made to work always using thunks

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 **withtype** and **cases**:

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

TVRCFAE Grammar

```
<TVRCFAE>  ::=  <num>
             |  {+ <TVRCFAE> <TVRCFAE>}
             |  {- <TVRCFAE> <TVRCFAE>}
             |  <id>
             |  {fun {<id>} <TVRCFAE>}
             |  {<TVRCFAE> <TVRCFAE>}
             |  {if0 <TVRCFAE> <TVRCFAE> <TVRCFAE>}
             |  {rec {<id> : <TE> <TVRCFAE>} <TVRCFAE>}
             |  {withtype {<tyid> {<id>  <TE>}                 NEW
                                           {<id>  <TE>}}
                          <TVRCFAE>}
             |  {cases <tyid> <TVRCFAE>                        NEW
                  {<id> {<id>} <TVRCFAE>}
                  {<id> {<id>} <TVRCFAE>}}

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

Well-Formed Type Expressions

- Might be ok:

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

- Not ok:

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

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

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

TVRCFAE Type Checker

$$\frac{\begin{array}{c}\Gamma'=\Gamma[\,\texttt{<tyid>} = \texttt{<id>}_1@\tau_1+\texttt{<id>}_2@\tau_2,\ \texttt{<id>}_1\leftarrow(\tau_1 \rightarrow \texttt{<tyid>}),\ \texttt{<id>}_2\leftarrow(\tau_2 \rightarrow \texttt{<tyid>})\,]\\ \Gamma' \vdash \tau_1 \qquad \Gamma' \vdash \tau_2 \qquad \Gamma' \vdash \texttt{e} : \tau_0\end{array}}{\Gamma \vdash \texttt{\{withtype \{<tyid> \{<id>}_1\ \ \tau_1\texttt{\} \{<id>}_2\ \ \tau_2\texttt{\}\} e\}} : \tau_0}$$

$$\frac{\begin{array}{c}\Gamma'=\Gamma[\,\texttt{<tyid>} = \texttt{<id>}_1@\tau_1+\texttt{<id>}_2@\tau_2\,]\\ \Gamma' \vdash \texttt{e}_0 : \texttt{<tyid>} \qquad \Gamma'[\,\texttt{<id>}_3\leftarrow\tau_1\,] \vdash \texttt{e}_1 : \tau_0 \qquad \Gamma'[\,\texttt{<id>}_4\leftarrow\tau_2\,] \vdash \texttt{e}_2 : \tau_0\end{array}}{\Gamma' \vdash \texttt{\{cases <tyid> e}_0\ \texttt{\{<id>}_1\ \texttt{\{<id>}_3\texttt{\} e}_1\texttt{\} \{<id>}_2\ \texttt{\{<id>}_4\texttt{\} e}_2\texttt{\}\}} : \tau_0}$$

Warning: later, we'll discuss why the **withtype** rule is not quite right

TVRCFAE Expression Datatypes

```
(define-type FAE
  [with-type (name : symbol)
             (var1-name : symbol)
             (var1-ty : TE)
             (var2-name : symbol)
             (var2-ty : TE)
             (body-expr : FAE)]
  [cases (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 (interp a-fae ds)
  (type-case FAE a-fae
    ...
    [with-type (type-name var1-name var1-te
                          var2-name var2-te
                          body-expr)
               (interp body-expr
                       (aSub var1-name
                             (constructorV false)
                             (aSub var2-name
                                   (constructorV true)
                                   ds)))]
    ...))
```

TVRCFAE Interpreter

```
(define (interp a-fae ds)
  (type-case FAE a-fae
    ...
    [app (fun-expr arg-expr)
         (local [(define fun-val
                   (interp fun-expr ds))
                 (define arg-val
                   (interp arg-expr ds))]
           (type-case FAE-Value fun-val
             [closureV (param body ds)
                       (interp body
                               (aSub param
                                     arg-val
                                     ds))]
             [constructorV (right?)
                           (variantV right? arg-val)]
             [else (error 'interp "not applicable")]))]
    ...))
```

TVRCFAE Interpreter

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

TVRCFAE Type Lookup

```
(define (get-type name-to-find env)
  (type-case TypeEnv env
    [mtEnv () (error 'get-type "free variable, so no type")]
    [aBind (name ty rest)
           (if (symbol=? name-to-find name)
               ty
               (get-type name-to-find rest))]
    [tBind (name var1-name var1-ty var2-name var2-ty rest)
           (get-type 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 (symbol=? 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 TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [fun (name te body)
           (local [(define arg-type (parse-type te))]
             (begin
               (validtype arg-type env)
               (arrowT arg-type
                       (typecheck body (aBind name
                                              arg-type
                                              env)))))]
      ...)))
```

TVRCFAE Type Checking

```
(define typecheck : (FAE TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [with-type (type-name var1-name var1-te var2-name var2-te
                            body-expr)
                 (local [(define var1-ty (parse-type var1-te))
                         (define var2-ty (parse-type var2-te))
                         (define 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 TypeEnv -> Type)
  (lambda (fae env)
    (type-case FAE fae
      ...
      [cases (type-name dispatch-expr
                        var1-name var1-id var1-rhs
                        var2-name var2-id var2-rhs)
          (local [(define bind (find-type-id type-name env))]
            (if (and (equal? var1-name (tBind-var1-name bind))
                     (equal? var2-name (tBind-var2-name bind)))
                (type-case Type (typecheck dispatch-expr env)
                  [idT (name)
                       (if (equal? name type-name)
                           (local [(define rhs1-ty
                                     (typecheck var1-rhs
                                                (aBind var1-id
                                                       (tBind-var1-type bind)
                                                       env)))
                                   (define rhs2-ty
                                     (typecheck var2-rhs
                                                (aBind var2-id
                                                       (tBind-var2-type bind)
                                                       env)))]
                             (if (equal? rhs1-ty rhs2-ty)
                                 rhs1-ty
                                 (type-error var2-rhs (to-string rhs1-ty))))
                           (type-error dispatch-expr (to-string type-name)))]
                  [else (type-error dispatch-expr (to-string type-name))])
                (type-error fae "matching variant names")))]
      ...)))
```