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> : <tyexp>} <TRCFAE>}
           |   {<TRCFAE> <TRCFAE>}
           |   {if0 <TRCFAE> <TRCFAE> <TRCFAE>}
           |   {rec {<id> : <tyexp> <TRCFAE>} <TRCFAE>}
<tyexp>   ::=  num
           |   (<tyexp> -> <tyexp>)
```

TRCFAE Datatypes

```
type fae =
    Num of int
  | Add of fae * fae
  | Sub of fae * fae
  | Id of string
  | Fun of string * te * fae
  | App of fae * fae
  | IfZ of fae * fae * fae
  | Rec of string * te * fae * fae

and subCache =
    MTSub
  | ASub of string * faeValue * subCache
  | ARecSub of string * faeValue ref * subCache
```

TRCFAE Interpreter

```
let rec lookup = function
    (findName, MTSub) -> raise (Failed "free variable")
  | (findName, ASub(name, v, restSc)) ->
      if (name = findName)
      then v
      else lookup(findName, restSc)
  | (findName, ARecSub(name, rv, restSc)) ->
      if (name = findName)
      then !rv
      else lookup(findName, restSc)
```

TRCFAE Interpreter

```
let rec interp : (fae * subCache -> faeValue ) = function
      …
  | (IfZ(tst, thn, els), sc) ->
      (match interp(tst, sc) with
        NumV(0) -> interp(thn, sc)
      | _ -> interp(els, sc))
      …
```

TRCFAE Interpreter

```
let rec interp : (fae * subCache -> faeValue ) = function
      …
  | (Rec(name, texpr, rhs, body), sc) ->
      let r = ref(NumV(0))
      in let sc = ARecSub(name, r, sc)
      in let v = interp(rhs, sc)
      in (r := v;
          interp(body, sc))
```

TRCFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    …
  | (IfZ(tst, thn, els), env) ->
      (match typecheck(tst, env) with
       NumT ->
         let thnType = typecheck(thn, env)
         and elsType = typecheck(els, env)
         in if (thnType = elsType)
         then thnType
         else raise (NoType (IfZ(tst, thn, els),
                             "results mismatch"))
      | _ -> raise (NoType (IfZ(tst, thn, els),
                            "test of non-number")))
    …
```

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

TRCFAE Type Checker

```
let rec typecheck : (fae * typeEnv -> ty) = function
    …
  | (Rec(name, texpr, rhs, body), env) ->
      let bindType = parseType(texpr)
      in let env = ABind(name, bindType, env)
      in if (typecheck(rhs, env) = bindType)
      then typecheck(body, env)
      else raise (NoType (Rec(name, texpr, rhs, body),
                              "bind mismatch"))
```

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

Pairs

```
{with {pair : (num -> (num -> (num -> num)))
              {fun {x : num}
                   {fun {y : num}
                        {fun {s : num}
                             {if0 s x y}}}}}
  {with {fst : ((num -> num) -> num)
              {fun {p : (num -> num)}
                   {p 0}}}
    {with {snd : ((num -> num) -> num)
                {fun {p : (num -> num)}
                     {p 1}}}
      {snd {{pair 1} 2}}}}}
```

Pairs

```
{with {pair : (bool -> (bool -> (num -> bool)))
              {fun {x : bool}
                   {fun {y : bool}
                        {fun {s : num}
                             {if0 s x y}}}}}
  {with {fst : ((num -> bool) -> bool)
              {fun {p : (num -> bool)}
                   {p 0}}}
    {with {snd : ((num -> bool) -> bool)
                {fun {p : (num -> bool)}
                     {p 1}}}
      {snd {{pair true} false}}}}}
```

Pairs

```
{with {pair : (num -> (bool -> (num -> …)))
              {fun {x : num}
                   {fun {y : bool}
                        {fun {s : num}
                             {if0 s x y}}}}}
  {with {fst : ((num -> …) -> …)
              {fun {p : (num -> …)}
                   {p 0}}}
    {with {snd : ((num -> …) -> …)
                {fun {p : (num -> …)}
                     {p 1}}}
      {snd {{pair 1} false}}}}}
```

Again, no possible type for …

TPRCFAE Grammar

```
<TPRCFAE>  ::=  <num>
            |   {+ <TPRCFAE> <TPRCFAE>}
            |   {- <TPRCFAE> <TPRCFAE>}
            |   <id>
            |   {fun {<id>} <TPRCFAE>}
            |   {<TPRCFAE> <TPRCFAE>}
            |   {if0 <TPRCFAE> <TPRCFAE> <TPRCFAE>}
            |   {rec {<id> : <tyexp> <TPRCFAE>} <TPRCFAE>}
            |   {pair <TPRCFAE> <TPRCFAE>}                    NEW
            |   {fst <TPRCFAE>}                               NEW
            |   {snd <TPRCFAE>}                               NEW
<tyexp>    ::=  num
            |   (<tyexp> -> <tyexp>)
            |   (<tyexp> * <tyexp>)                           NEW
```

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

TPRCFAE Grammar

```
<TPRCFAE>  ::=  <num>
            |   {+ <TPRCFAE> <TPRCFAE>}
            |   {- <TPRCFAE> <TPRCFAE>}
            |   <id>
            |   {fun {<id>} <TPRCFAE>}
            |   {<TPRCFAE> <TPRCFAE>}
            |   {if0 <TPRCFAE> <TPRCFAE> <TPRCFAE>}
            |   {rec {<id> : <tyexp> <TPRCFAE>} <TPRCFAE>}
            |   {pair <TPRCFAE> <TPRCFAE>}                    NEW
            |   {fst <TPRCFAE>}                               NEW
            |   {snd <TPRCFAE>}                               NEW
<tyexp>    ::=  num
            |   (<tyexp> -> <tyexp>)
            |   (<tyexp> * <tyexp>)                           NEW
```

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

TPRCFAE Grammar

```
<TPRCFAE>  ::=  <num>
            |   {+ <TPRCFAE> <TPRCFAE>}
            |   {- <TPRCFAE> <TPRCFAE>}
            |   <id>
            |   {fun {<id>} <TPRCFAE>}
            |   {<TPRCFAE> <TPRCFAE>}
            |   {if0 <TPRCFAE> <TPRCFAE> <TPRCFAE>}
            |   {rec {<id> : <tyexp> <TPRCFAE>} <TPRCFAE>}
            |   {pair <TPRCFAE> <TPRCFAE>}                    NEW
            |   {fst <TPRCFAE>}                               NEW
            |   {snd <TPRCFAE>}                               NEW
<tyexp>    ::=  num
            |   (<tyexp> -> <tyexp>)
            |   (<tyexp> * <tyexp>)                           NEW
```

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

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 {intlist {empty int}
                   {cons (int * intlist)}}
  {rec {len : (intlist -> int)
             {fun {l : intline}
                  {cases intlist 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> : <tyexp> <TVRCFAE>} <TVRCFAE>}
            |   {withtype {<tyid> {<id>  <tyexp>}                NEW
                                  {<id>  <tyexp>}}
                  <TVRCFAE>}
            |   {cases <tyid> <TVRCFAE>                          NEW
                  {<id> {<id>} <TVRCFAE>}
                  {<id> {<id>} <TVRCFAE>}}
<tyexp>    ::=  num
            |   (<tyexp> -> <tyexp>)
            |   <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

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

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

Warning: next time, we'll discuss why the `withtype` rule is not quite right

TVRCFAE Datatypes

```
type fae =
     …
   | WithType of string * string * te * string * te * fae
   | Cases of string*fae * string*string*fae * string*string*fae

and te =
     …
   | IdTE of string

and faeValue =
     …
   | VariantV of bool * faeValue
   | ConstructorV of bool


…
and ty =
     …
   | IdT of string

and typeEnv =
     …
   | TBind of string * string * ty * string * ty * typeEnv
```

TVRCFAE Interpreter

```
let rec interp : (fae * subCache -> faeValue ) = function
    …
  | (WithType(tyname, v1name, _, v2name, _, body), sc) ->
      interp(body, ASub(v1name, ConstructorV(false),
                        ASub(v2name, ConstructorV(true),
                             sc)))
    …
```

TVRCFAE Interpreter

```
let rec interp : (fae * subCache -> faeValue ) = function
    …
  | (App(fn, arg), sc) ->
      let funV = interp(fn, sc)
      and argV = interp(arg, sc)
      in (match funV with
        ClosureV(param, body, sc) ->
          interp(body, ASub(param, argV, sc))
      | ConstructorV(isright) ->
          VariantV(isright, argV)
      | _ -> raise (Failed "not a function"))
    …
```

TVRCFAE Interpreter

```
let rec interp : (fae * subCache -> faeValue ) = function
    …
  | (Cases(tyname, tst, _, id1, body1, _, id2, body2), sc)  ->
      (match interp(tst, sc) with
        VariantV(false, v) -> interp(body1, ASub(id1, v, sc))
      | VariantV(true, v) -> interp(body2, ASub(id2, v, sc))
      | _ -> raise (Failed "not a variant"))
```

TVRCFAE Type Lookup

```
let rec gettype = function
    (findName, MTEnv) -> raise (NoType (Id(findName),
                                        "free variable"))
  | (findName, ABind(name, t, restEnv)) ->
      if (name = findName)
      then t
      else gettype(findName, restEnv)
  | (findName, TBind(name, v1, t1, v2, t2, restEnv)) ->
      gettype(findName, restEnv)
```

TVRCFAE Type Lookup

```
exception NoTypeName of string

let rec findtypeid = function
    (findName, env) -> match env with
      MTEnv -> raise (NoTypeName findName)
    | ABind(name, t, restEnv) ->
        findtypeid(findName, restEnv)
    | TBind(name, v1, t1, v2, t2, restEnv) ->
        if (name = findName)
        then env
        else findtypeid(findName, restEnv)
```

TVRCFAE Type-Expression Checking

```
let rec validtype = function
    (t, env) -> match t with
      NumT -> ()
    | BoolT -> ()
    | ArrowT(a, b) -> (validtype(a, env); validtype(b, env))
    | IdT(name) -> let env = findtypeid(name, env) in ()
```

TVRCFAE Type Checking

```
let rec typecheck : (fae * typeEnv -> ty) = function
    (expr, env) -> match expr with
      Num(n) -> NumT
      …
    | Fun(param, texpr, body) ->
        let argType = parseType(texpr)
        in (validtype(argType, env);
            ArrowT(argType, typecheck(body, ABind(param, argType, env))))
      …
```

TVRCFAE Type Checking

```
let rec typecheck : (fae * typeEnv -> ty) = function
    (expr, env) -> match expr with
      …
    | WithType(tname,v1name,te1,v2name,te2,body) ->
        let t1 = parseType(te1)
        and t2 = parseType(te2)
        in let env = TBind(tname, v1name, t1, v2name, t2, env)
        in (validtype(t1, env);
            validtype(t2, env);
            typecheck(body, ABind(v1name, ArrowT(t1, IdT(tname)),
                                  ABind(v2name, ArrowT(t2, IdT(tname)),
                                        env))))
```

TVRCFAE Type Checking

```
let rec typecheck : (fae * typeEnv -> ty) = function
    (expr, env) -> match expr with
      ...
    | Cases(tname, vexpr, v1name, id1, body1, v2name, id2, body2) ->
        let binding = findtypeid(tname, env)
        in if (typecheck(vexpr, env) = IdT(tname))
        then (match binding with
          TBind(name, v1n, t1, v2n, t2, _) ->
            if ((v1n = v1name) & (v2n = v2name))
            then (let b1t = typecheck(body1, ABind(id1, t1, env))
                  and b2t = typecheck(body2, ABind(id2, t2, env))
                  in if (b1t == b2t)
                  then b1t
                  else raise (NoType (expr, "branch mismatch")))
            else raise (NoType (expr, "wrong variant names"))
        | _ -> raise (Failed "can't happen"))
        else raise (NoType (expr, "value type mismatch"))
```