Part I

Recursion

```
let mk_rec = (fun (body):
                (fun (fX):
                   fX(fX))(fun (fX):
                              (fun (f):
                                 body(f))(fun (x):
                                            fX(fX)(x)))):
  let fib = mk_rec(fun (fib):
                     fun (n):
                       if n == 0
                       | 1
                       | if n - 1 == 0
                         | 1
                         | fib(n - 1)
                             + fib(n - 2)):
    fib(4)
```

Typed Recursion

```
let mk_rec :: .... = (fun (body :: (Int -> Int) -> Int -> Int):
                        (fun (fX :: … -> Int -> Int):
                           fX(fX))(fun (fX :: … -> Int -> Int):
                                     (fun (f :: Int -> Int):
                                        body(f))(fun (x :: Int):
                                                   fX(fX)(x)))):
  let fib :: Int -> Int = mk_rec(fun (fib :: Int -> Int):
                                   fun (n :: Int):
                                     if n == 0
                                     | 1
                                     | if n - 1 == 0
                                       | 1
                                       | fib(n - 1)
                                           + fib(n - 2)):
    fib(4)
```

Nothing works in place of …

Typed Recursion

```
let mk_rec :: .... = (fun (body :: (Int -> Int) -> Int -> Int):
                        (fun (fX :: … -> Int -> Int):
                           fX(fX))(fun (fX :: … -> Int -> Int):
                                     (fun (f :: Int -> Int):
                                        body(f))(fun (x :: Int):
                                                   fX(fX)(x)))):
  let fib :: Int -> Int = mk_rec(fun (fib :: Int -> Int):
                                   fun (n :: Int):
                                     if n == 0
                                     | 1
                                     | if n - 1 == 0
                                       | 1
                                       | fib(n - 1)
                                           + fib(n - 2)):
    fib(4)
```

Theorem: programs in the *simply typed λ-calculus* always terminate

Extending the Type System

When encodings fail, extend the language and type system

In this case, add **`letrec`** as a core form, again

```
letrec fib :: Int -> Int = (fun (n :: Int):
                              if n == 0
                              | 1
                              | if n - 1 == 0
                                | 1
                                | fib(n - 1)
                                    + fib(n - 2)):
  fib(4)
```

Grammar with Recursion

```
<Exp>    ::=  <Int>
           |  <Exp> + <Exp>
           |  <Exp> * <Exp>
           |  <Symbol>
           |  fun (<Symbol> :: <Type>): <Exp>
           |  <Exp>(<Exp>)
           |  (<Exp>)
           |  letrec <Symbol> :: <Type> = <Exp>: <Exp>

<Type>   ::=  Int
           |  Boolean
           |  <Type> -> <Type>
           |  (<Type>)
```

Datatypes

```
type Exp
...
| letrecE(name :: Symbol,
          rhs_type :: Type,
          rhs :: Exp,
          body :: Exp)


type Binding
| bind(name :: Symbol,
       val :: Boxof(Optionof(Value)))
```

Interpreter with Recursion

```
fun interp(a, env):
  match env
  | ...
  | letrecE(n, t, rhs, body):
      def b = box(none())
      def new_env = extend_env(bind(n, b),
                               env)
      set_box(b, some(interp(rhs, new_env)))
      interp(body, new_env)
```

Type Checking Recursion

$$\frac{\Gamma[\,\mathbf{x}\leftarrow\tau_0\,] \vdash \mathbf{e}_0 \;:\; \tau_0 \qquad \Gamma[\,\mathbf{x}\leftarrow\tau_0\,] \vdash \mathbf{e}_1 \;:\; \tau_1}{\Gamma \vdash \texttt{letrec}\ \mathbf{x}\ \texttt{::}\ \tau_0\ \texttt{=}\ \mathbf{e}_0\texttt{:}\ \mathbf{e}_1 \;:\; \tau_1}$$

Abbreviation: **x** = **<Symbol>**

Type Checker with Recursion

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, env):
    match a
    | ...
    | letrecE(n, rhs_type, rhs, body):
        ....
```

$$\frac{\Gamma[\,\mathbf{x}\leftarrow\tau_0\,] \vdash \mathbf{e}_0 \;:\; \tau_0 \qquad \Gamma[\,\mathbf{x}\leftarrow\tau_0\,] \vdash \mathbf{e}_1 \;:\; \tau_1}{\Gamma \vdash \mathbf{letrec}\ \mathbf{x}\ ::\ \tau_0\ =\ \mathbf{e}_0 :\ \mathbf{e}_1 \;:\; \tau_1}$$

Type Checker with Recursion

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, env):
    match a
    | ...
    | letrecE(n, rhs_type, rhs, body):
        ....
        typecheck(rhs, ....)
        typecheck(body, ....)
        ....
```

$$\frac{\Gamma[\,\mathbf{x}\leftarrow\tau_0\,] \vdash \mathbf{e}_0 \;:\; \tau_0 \qquad \Gamma[\,\mathbf{x}\leftarrow\tau_0\,] \vdash \mathbf{e}_1 \;:\; \tau_1}{\Gamma \vdash \mathtt{letrec}\ \mathbf{x}\ \mathtt{::}\ \tau_0\ \mathtt{=}\ \mathbf{e}_0\mathtt{:}\ \mathbf{e}_1 \;:\; \tau_1}$$

Type Checker with Recursion

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, env):
    match a
    | ...
    | letrecE(n, rhs_type, rhs, body):
        def new_tenv = extend_env(tbind(n, rhs_type),
                                  tenv)
        ....
        typecheck(rhs, new_tenv)
        typecheck(body, new_tenv)
        ....
```

$$\frac{\Gamma[\,\mathbf{x}\leftarrow\tau_0\,] \vdash \mathbf{e}_0 : \tau_0 \qquad \Gamma[\,\mathbf{x}\leftarrow\tau_0\,] \vdash \mathbf{e}_1 : \tau_1}{\Gamma \vdash \mathtt{letrec}\ \mathbf{x} :: \tau_0 = \mathbf{e}_0 : \mathbf{e}_1 : \tau_1}$$

Type Checker with Recursion

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, env):
    match a
    | ...
    | letrecE(n, rhs_type, rhs, body):
        def new_tenv = extend_env(tbind(n, rhs_type),
                                  tenv)
        if rhs_type == typecheck(rhs, new_tenv)
        | typecheck(body, new_tenv)
        | type_error(rhs, to_string(rhs_type))
```

$$\frac{\Gamma[\,\mathbf{x}\leftarrow\tau_0\,] \vdash \mathbf{e}_0 : \tau_0 \qquad \Gamma[\,\mathbf{x}\leftarrow\tau_0\,] \vdash \mathbf{e}_1 : \tau_1}{\Gamma \vdash \mathtt{letrec}\ \mathbf{x} :: \tau_0 = \mathbf{e}_0 : \mathbf{e}_1 : \tau_1}$$

Part 2

Forms of Data

- Atomic
  - numbers
  - booleans
  - ...
- Compound
  - Tuples
  - Records
  - ...
- Variants

Variants

```
type Grade
| percent(r :: Int)
| passfail(is_pass :: Boolean)

type Chain
| end(r :: Int)
| link(next :: Chain)

type IntList
| iempty(d :: Int)
| icons(p :: Int * IntList)

...
```

Variants

```
let percent :: Int -> Int * (Int * Boolean) = (fun (x :: Int):
                                                pair(0, pair(x, #false))):
  let passfail :: Boolean -> Int * (Int * Boolean) = (fun (y :: Boolean):
                                                       pair(1, pair(0, y))):
    let is_pass :: Int * (Int * Boolean) -> Boolean = (fun (p :: Int * (Int * Boolean)):
                                                         if fst(p) == 0
                                                         | fst(snd(p)) > 70
                                                         | snd(snd(p))):
      is_pass(percent(96))
```

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

Recursive Datatypes

```
let end :: Int -> … = (fun (d :: Int):
                         pair(0, d)):
  let link :: … -> … = (fun (r :: …):
                          pair(1, r)):
    link(link(link(end(0))))
```

Recursive Datatypes

```
let end :: Int -> Int * Int = (fun (d :: Int):
                                 pair(0, d)):
  let link :: … -> … = (fun (r :: …):
                          pair(1, r)):
    link(link(link(end(0))))
```

Recursive Datatypes

```
let end :: Int -> Int * Int = (fun (d :: Int):
                                 pair(0, d)):
  let link :: Int * Int -> … = (fun (r :: …):
                                  pair(1, r)):
    link(link(link(end(0))))
```

Recursive Datatypes

```
let end :: Int -> Int * Int = (fun (d :: Int):
                                 pair(0, d)):
  let link :: Int * Int -> … = (fun (r :: Int * Int):
                                  pair(1, r)):
    link(link(link(end(0))))
```

Recursive Datatypes

```
let end :: Int -> Int * Int = (fun (d :: Int):
                                 pair(0, d)):
  let link :: Int * Int -> Int * (Int * Int) = (fun (r :: Int * Int):
                                                  pair(1, r)):
    link(link(link(end(0))))
```

Stuck...

Recursive Datatypes

```
let iempty :: Int -> Int * … = (fun (d :: Int):
                                  pair(0, d)):
  let icons :: Int -> Int * … -> Int * … = (fun (x :: Int):
                                             fun (r :: Int * …):
                                               pair(1, pair(x, r))):
    icons(1)(icons(2)(icons(3)(iempty(0))))
```

Stuck again with …

Recursive Datatypes

Add **`let_type`** and **`match`**:

```
let_type (Intlist
          | iempty(arg :: Int)
          | icons(arg :: Int * Intlist)):
  letrec len :: Intlist -> Int = (fun (l :: Intlist):
                                    match l :: Intlist
                                    | iempty(n): 0
                                    | icons(fxr): 1 + len(snd(fxr))):
    len(icons(pair(1, icons(pair(2, iempty(0))))))
```

Grammar with Variants

```
<Exp>   ::= <Int>
          | <Exp> + <Exp>
          | <Exp> - <Exp>
          | <Symbol>
          | fun (<Symbol> :: <Type>): <Exp>
          | <Exp>(<Exp>)
          | (<Exp>)
          | letrec <Symbol> :: <Type> = <Exp>: <Exp>
          | let_type (<Symbol> | <Symbol>(arg :: <Type>)
                               | <Symbol>(arg :: <Type>)):
              <Exp>
          | match <Exp> :: <Symbol>
            | <Symbol>(<Symbol>): <Exp>
            | <Symbol>(<Symbol>): <Exp>

<Type>  ::= Int
          | Boolean
          | <Type> -> <Type>
          | (<Type>)
          | <Symbol>
```

Part 3

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                           ...'),
               mt_env)
       ~is ...
```

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                        1'),
               mt_env)
       ~is intV(1)
```

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                        apple(1)'),
               mt_env)
       ~is ....
```

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                        apple(1)'),
               mt_env)
       ~is appleV(intV(1)) // ???
```

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                        apple(1)'),
               mt_env)
       ~is variantV(#'apple, intV(1))
```

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                        banana(fun (x :: Int): 5)'),
               mt_env)
       ~is variantV(#'banana, closV(...))
```

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                          apple'),
               mt_env)
       ~is ....
```

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                        apple'),
               mt_env)
       ~is constructorV(#'apple)
```

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                        banana'),
               mt_env)
       ~is constructorV(#'banana)
```

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                        match apple(3) :: Fruit
                        | apple(a): a
                        | banana(b): b(7)'),
               mt_env)
       ~is intV(3)
```

Examples

```
check: interp(parse('let_type (Fruit | apple(arg :: Int)
                                     | banana(arg :: Int -> Int)):
                       match banana(fun (x :: Int):
                                      x + 3) :: Fruit
                        | apple(a): a
                        | banana(b): b(7)'),
               mt_env)
       ~is intV(10)
```

Part 4

Value Datatype

```
type Value
....
| variantV(tag :: Symbol,
           val :: Value)
| constructorV(tag :: Symbol)
```

```
constructorV(#'apple)
```

Value Datatype

```
type Value
....
| variantV(tag :: Symbol,
           val :: Value)
| constructorV(tag :: Symbol)
```

Apply `constructorV(#'apple)` to `intV(1)`

⇒ `variantV(#'apple, intV(1))`

Expressions for Variants

```
type Exp
....
| let_typeE(type_name :: Symbol,
            tag1 :: Symbol,
            type1 :: Type,
            tag2 :: Symbol,
            type2 :: Type,
            body :: Exp)
| matchE(tst :: Exp,
         type_name :: Symbol,
         tag1 :: Symbol,
         n1 :: Symbol,
         rhs1 :: Exp,
         tag2 :: Symbol,
         n2 :: Symbol,
         rhs2 :: Exp)

type Type
....
| definedT(name :: Symbol)
```

Expressions for Variants

```
type Exp
....
| let_typeE(type_name :: Symbol,
             tag1 :: Symbol,
             type1 :: Type,
             tag2 :: Symbol,
             type2 :: Type,
             body :: Exp)
| matchE(tst :: Exp,
          type_name :: Symbol,
          tag1 :: Symbol,
          n1 :: Symbol,
          rhs1 :: Exp,
          tag2 :: Symbol,
          n2 :: Symbol,
          rhs2 :: Exp)

type Type
....
| definedT(name :: Symbol)
```

```
let_type (<Symbol> | <Symbol>(arg :: <Type>)
                   | <Symbol>(arg :: <Type>)):
  <Exp>
```

Expressions for Variants

```
type Exp
....
| let_typeE(type_name :: Symbol,
            tag1 :: Symbol,
            type1 :: Type,
            tag2 :: Symbol,
            type2 :: Type,
            body :: Exp)
| matchE(tst :: Exp,
         type_name :: Symbol,
         tag1 :: Symbol,
         n1 :: Symbol,
         rhs1 :: Exp,
         tag2 :: Symbol,
         n2 :: Symbol,
         rhs2 :: Exp)

type Type
....
| definedT(name :: Symbol)
```

```
match <Exp> :: <Symbol>
| <Symbol>(<Symbol>): <Exp>
| <Symbol>(<Symbol>): <Exp>
```

Expressions for Variants

```
type Exp
....
| let_typeE(type_name :: Symbol,
            tag1 :: Symbol,
            type1 :: Type,
            tag2 :: Symbol,
            type2 :: Type,
            body :: Exp)
| matchE(tst :: Exp,
          type_name :: Symbol,
          tag1 :: Symbol,
          n1 :: Symbol,
          rhs1 :: Exp,
          tag2 :: Symbol,
          n2 :: Symbol,
          rhs2 :: Exp)

type Type
....
| definedT(name :: Symbol)
```

Interpreter

```
fun interp(a, tenv):
  match a
  | ....
  | let_typeE(type_name,
              tag1, type1,
              tag2, type2,
              body):
      interp(body,
             extend_env(bind(tag1,
                             box(some(constructorV(tag1)))),
                        extend_env(bind(tag2,
                                        box(some(constructorV(tag2)))),
                                   env)))
  | ....
```

Interpreter

```
fun interp(a, tenv):
  match a
  | ....
  | let_typeE(type_name,
              tag1, type1,
              tag2, type2,
              body):
      interp(body,
             extend_env(bind(tag1,
                             box(some(constructorV(tag1)))),
                        extend_env(bind(tag2,
                                        box(some(constructorV(tag2)))),
                                   env)))
  | ....
```

```
let_type (Fruit | apple(arg :: Int)
                | banana(arg :: Int -> Int)):
  apple(10)
```

Interpreter

```
fun interp(a, tenv):
  match a
  | ....
  | appE(fn, arg):
      match interp(fn, env)
      | closV(n, body, c_env): ....
      | constructorV(tag):
          variantV(tag, interp(arg, env))
      | ~else: error(#'interp, "not a function")
  | ....
```

Interpreter

```
fun interp(a, tenv):
  match a
  | ....
  | appE(fn, arg):
      match interp(fn, env)
      | closV(n, body, c_env): ....
      | constructorV(tag):
          variantV(tag, interp(arg, env))
      | ~else: error(#'interp, "not a function")
  | ....
```

```
let_type (Fruit | apple(arg :: Int)
                | banana(arg :: Int -> Int)):
  apple(10)
```

Interpreter

```
fun interp(a, tenv):
  match a
  | ....
  | matchE(tst, type_name,
           tag1, n1, rhs1,
           tag2, n2, rhs2):
      match interp(tst, env)
      | variantV(tag, val):
          cond
          | tag == tag1: interp(rhs1,
                                extend_env(bind(n1, box(some(val))),
                                           env))
          | tag == tag2: interp(rhs2,
                                extend_env(bind(n2, box(some(val))),
                                           env))
          | ~else: error(#'interp, "wrong tag")
      | ~else: error(#'interp, "not a variant")
  | ....
```

Interpreter

```
fun interp(a, tenv):
  match a
  | ....
  | matchE(tst, type_name,
           tag1, n1, rhs1,
           tag2, n2, rhs2):
      match interp(tst, env)
      | variantV(tag, val):
          cond
          | tag == tag1: interp(rhs1,
                                extend_env(bind(n1, box(some(val))),
                                           env))
          | tag == tag2: interp(rhs2,
                                extend_env(bind(n2, box(some(val))),
                                           env))
          | ~else: error(#'interp, "wrong tag")
      | ~else: error(#'interp, "not a variant")
  | ....
```

```
match apple(10) :: Fruit
| apple(a): a
| banana(b): b(7)
```

Part 5

Typechecking Examples

```
check: typecheck(parse('let_type (Fruit | apple(arg :: Int)
                                        | banana(arg :: Int -> Int)):
                                  ...'),
                       mt_env)
       ~is ...
```

Typechecking Examples

```
check: typecheck(parse('let_type (Fruit | apple(arg :: Int)
                                        | banana(arg :: Int -> Int)):
                          1'),
                 mt_env)
       ~is intT()
```

Typechecking Examples

```
check: typecheck(parse('let_type (Fruit | apple(arg :: Int)
                                        | banana(arg :: Int -> Int)):
                          apple(1)'),
                 mt_env)
       ~is definedT(#'Fruit)
```

Typechecking Examples

```
check: typecheck(parse('let_type (Fruit | apple(arg :: Int)
                                        | banana(arg :: Int -> Int)):
                          banana(fun (x :: Int): x)'),
                 mt_env)
       ~is definedT(#'Fruit)
```

Typechecking Examples

```
check: typecheck(parse('let_type (Fruit | apple(arg :: Int)
                                        | banana(arg :: Int -> Int)):
                          match apple(3) :: Fruit
                          | apple(a): a
                          | banana(b): b(7)'),
                 mt_env)
       ~is intT()
```

`match` results must all have the same type

Typechecking Examples

```
check: typecheck(parse('let_type (Fruit | apple(arg :: Int)
                                        | banana(arg :: Int -> Int)):
                          match apple(3) :: Fruit
                          | radish(a): a
                          | banana(b): b(7)'),
                 mt_env)
       ~raises "no type"
```

`typecheck` must record variant names for `let_type`

Typechecking Examples

```
check: typecheck(parse('let_type (Fruit | apple(arg :: Int)
                                        | banana(arg :: Int -> Int)):
                          match apple(3) :: Fruit
                          | apple(a): a
                          | banana(b): b'),
                 mt_env)
       ~raises "no type"
```

`typecheck` must record variant types for `let_type`

Part 6

Bindings for Defined Types

```
let_type (D | x1(arg :: τ1)
            | x2(arg :: τ2)):
  ...
```

$\Gamma[\ \mathbf{D} = \mathbf{x}_1@\tau_1 + \mathbf{x}_2@\tau_2\ ]$

Bindings for Defined Types

$\Gamma[\ \mathbf{D} = \mathbf{x}_1@\tau_1 + \mathbf{x}_2@\tau_2\ ]$

```
type Type-Binding
....
| tdef(type_name :: Symbol,
       tag1 :: Symbol,
       type1 :: Type,
       tag2 :: Symbol,
       type2 :: Type)
```

Local Type Names

- Might be ok:

```
let_type (Fruit | apple(arg :: Int)
                | banana(arg :: Int -> Int)):
  ... fun (x :: Fruit): ... ...
```

- Not ok:

```
fun (x :: Fruit): ...
```

$$[\ \ldots\ \mathbf{D} = \mathbf{x}_1@\tau_1 + \mathbf{x}_2@\tau_2\ \ \ldots\ ] \vdash \mathbf{D}$$

$$\Gamma \vdash \texttt{Int} \qquad \Gamma \vdash \texttt{Boolean} \qquad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1\ \texttt{->}\ \tau_2}$$

Local Type Names

- Might be ok:

```
let_type (Fruit | apple(arg :: Int)
                | banana(arg :: Int -> Int)):
  ... fun (x :: Fruit): ... ...
```

- Not ok:

```
fun (x :: Fruit):
```

tvarcheck

$$[\ \ldots\ \mathbf{D} = \mathbf{x}_1@\tau_1 + \mathbf{x}_2@\tau_2\ \ \ldots\ ]\ \vdash\ \mathbf{D}$$

$$\Gamma \vdash \texttt{Int} \qquad \Gamma \vdash \texttt{Boolean} \qquad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1\ \texttt{->}\ \tau_2}$$

Part 7

Type Checking

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma[\texttt{<Symbol>} \leftarrow \tau_1] \vdash \texttt{e} : \tau_2}{\Gamma \vdash \texttt{fun (<Symbol> :: } \tau_1 \texttt{) : e : } \tau_1 \texttt{ -> } \tau_2}$$

```
fun typecheck(a, tenv):
  match a
  | ....
  | funE(n, arg_type, body):
      tvarcheck(arg_type, tenv)
      arrowT(arg_type,
             typecheck(body,
                       extend_env(tbind(n, arg_type),
                                  tenv)))
  | ....
```

Type Checker with Variants

$$\frac{\Gamma' = \Gamma[\ \mathbf{D} = \mathbf{x}_1@\tau_1 + \mathbf{x}_2@\tau_2,\ \mathbf{x}_1 \leftarrow \tau_1 \rightarrow \mathbf{D},\ \mathbf{x}_2 \leftarrow \tau_2 \rightarrow \mathbf{D}\ ] \qquad \Gamma' \vdash \tau_1 \qquad \Gamma' \vdash \tau_2 \qquad \Gamma' \vdash \mathbf{e} : \tau_0}{\Gamma \vdash \texttt{let\_type}\ (\mathbf{D} \mid \mathbf{x}_1(\texttt{arg} :: \tau_1) \mid \mathbf{x}_2(\texttt{arg} :: \tau_2)) : \mathbf{e} : \tau_0}$$

$$\frac{\Gamma = [\ \ldots\ \mathbf{D} = \mathbf{x}_1@\tau_1 + \mathbf{x}_2@\tau_2\ \ldots\ ] \qquad \Gamma \vdash \mathbf{e}_0 : \mathbf{D} \qquad \Gamma[\ \mathbf{x}_3 \leftarrow \tau_1\ ] \vdash \mathbf{e}_1 : \tau_0 \qquad \Gamma[\ \mathbf{x}_4 \leftarrow \tau_2\ ] \vdash \mathbf{e}_2 : \tau_0}{\Gamma \vdash \texttt{match}\ \mathbf{e}_0 :: \mathbf{D} \mid \mathbf{x}_1(\mathbf{x}_3) : \mathbf{e}_1 \mid \mathbf{x}_2(\mathbf{x}_4) : \mathbf{e}_2 : \tau_0}$$

Warning: this `let_type` rule is not quite right...

Type Checking

```
fun typecheck(a, tenv):
  match a
  | ....
  | let_typeE(type_name,
              tag1, type1,
              tag2, type2,
              body):
      def new_tenv = extend_env(tdef(type_name,
                                     tag1, type1,
                                     tag2, type2),
                                tenv)
      tvarcheck(type1, new_tenv)
      tvarcheck(type2, new_tenv)
      typecheck(body, extend_env(tbind(tag1,
                                       arrowT(type1, definedT(type_name))),
                                 extend_env(tbind(tag2,
                                                  arrowT(type2, definedT(type_name))),
                                            new_tenv)))
  | ....
```

Type Checking

```
fun typecheck(a, tenv):
  match a
  | ....
  | let_typeE(type_name,
              tag1, type1,
              tag2, type2,
              body):
      def new_tenv = extend_env(tdef(type_name,
                                     tag1, type1,
                                     tag2, type2),
                                tenv)
      tvarcheck(type1, new_tenv)
      tvarcheck(type2, new_tenv)
      typecheck(body, extend_env(tbind(tag1,
                                       arrowT(type1, definedT(type_name))),
                                 extend_env(tbind(tag2,
                                                  arrowT(type2, definedT(type_name))),
                                            new_tenv)))
  | ....
```

```
let_type (Fruit | apple(arg :: Int)
                | banana(arg :: Int -> Int)):
  apple(10)
```

Type Checking

```
fun typecheck(a, tenv):
  match a
  | ....
  | matchE(tst, type_name,
           tag1, n1, rhs1,
           tag2, n2, rhs2):
      def tdef = defined_type_lookup(type_name, tenv)
      ....
      match typecheck(tst, tenv)
      | definedT(name):
          .... name != type_name
          .... typecheck(rhs1, extend_env(tbind(n1, tdef_type1(tdef)),
                                          tenv))
          .... typecheck(rhs2, extend_env(tbind(n2, tdef_type2(tdef)),
                                          tenv))
          ....
      ....
  | ....
```

Type Checking

```
fun typecheck(a, tenv):
  match a
  | ....
  | matchE(tst, type_name,
           tag1, n1, rhs1,
           tag2, n2, rhs2):
      def tdef = defined_type_lookup(type_name, tenv)
      ....
      match typecheck(tst, tenv)
      | definedT(name):
          .... name != type_name
          .... typecheck(rhs1, extend_env(tbind(n1, tdef_type1(tdef)),
                                          tenv))
          .... typecheck(rhs2, extend_env(tbind(n2, tdef_type2(tdef)),
                                          tenv))
          ....
      ....
  | ....
```

```
match apple(10) :: Fruit
| apple(a): a
| banana(b): b(7)
```

Type Checking

```
fun typecheck(a, tenv):
  match a
  | ....
  | matchE(tst, type_name,
           tag1, n1, rhs1,
           tag2, n2, rhs2):
      def tdef = defined_type_lookup(type_name, tenv)
      if (tag1 != tdef_tag1(tdef)) || (tag2 != tdef_tag2(tdef))
      | type_error(a, "matching variant names")
      | match typecheck(tst, tenv)
        | definedT(name):
            if name != type_name
            | type_error(tst, to_string(type_name))
            | block:
                def rhs1_t = typecheck(rhs1, extend_env(tbind(n1, tdef_type1(tdef)),
                                                        tenv))
                def rhs2_t =  typecheck(rhs2, extend_env(tbind(n2, tdef_type2(tdef)),
                                                         tenv))
                if rhs1_t == rhs2_t
                | rhs1_t
                | type_error(rhs2, to_string(rhs1_t))
        | ~else: type_error(tst, to_string(type_name))
  | ....
```

Checking Type Forms

```
fun tvarcheck(ty, tenv):
  match ty
  | intT(): #void
  | boolT(): #void
  | arrowT(a, b):
      tvarcheck(a, tenv)
      tvarcheck(b, tenv)
  | definedT(id):
      begin:
        defined_type_lookup(id, tenv)
        #void
```

Part 8

Type Soundness

Type soundness is a theorem of the form

> If $\varnothing \vdash \mathbf{e} : \tau$, then running $\mathbf{e}$ never produces an error

If we add division, then divide-by-zero errors may be ok:

> If $\varnothing \vdash \mathbf{e} : \tau$, then running $\mathbf{e}$ 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

`letrec` checking has a bug:

```
letrec f :: Int -> Int = f:
  f(10)
```

Solution 1: change the grammar for `letrec`

```
<Exp>  ::=  ...
       |    letrec <Symbol> :: <Type> = (fun (<Symbol> :: <Type>):
                                           <Exp>):
              <Exp>
```

Solution 2: adjust the soundness theorem to allow a run-time error

Type Soundness

let_type checking has a bug, too:

```
let f = (let_type (Foo | a(arg :: Int) | b(arg :: Int)):
           fun (x :: Foo): 1 + match x :: Foo
                               | a(n): n
                               | b(n): n):
  f(let_type (Foo | a(arg :: Int -> Int) | b(arg :: Int)):
      a(fun (y :: Int): y))
```

Solution 1: no local type declarations

Solution 2: don't let **D** escape **let_type**

$$\frac{\Gamma' = \Gamma[\ \mathbf{D} = \mathbf{x}_1@\tau_1 + \mathbf{x}_2@\tau_2,\ \mathbf{x}_1 \leftarrow \tau_1 \rightarrow \mathbf{D},\ \mathbf{x}_2 \leftarrow \tau_2 \rightarrow \mathbf{D}\ ] \quad \Gamma' \vdash \tau_1 \quad \Gamma' \vdash \tau_2 \quad \Gamma' \vdash \mathbf{e} : \tau_0 \quad \mathbf{D} \text{ not in } \tau_0}{\Gamma \vdash \texttt{let\_type}\ (\mathbf{D} \mid \mathbf{x}_1(\texttt{arg} :: \tau_1) \mid \mathbf{x}_2(\texttt{arg} :: \tau_2)) : \mathbf{e} : \tau_0}$$