Part I

Recursion

```
{let {[mk-rec {lambda {body}
                {{lambda {fX} {fX fX}}
                 {lambda {fX}
                   {{lambda {f} {body f}}
                    {lambda {x} {{fX fX} x}}}}}}]}
  {let {[fib {mk-rec
              {lambda {fib}
                {lambda {n}
                  {if0 n
                       1
                       {if0 {- n 1}
                            1
                            {+ {fib {- n 1}}
                               {fib {- n 2}}}}}}}}]}
    {fib 4}}}
```

Typed Recursion

```
{let {[mk-rec : (((num -> num) -> (num -> num)) -> (num -> num))
               {lambda {[body : ((num -> num) -> (num -> num))]}
                 {{lambda {[fX : … -> (num -> num)]} {fX fX}}
                  {lambda {[fX : … -> (num -> num)]}
                    {{lambda {[f : (num -> num)]} {body f}}
                     {lambda {[x : num]} {{fX fX} x}}}}}}]}
  {let {[fib : (num -> num)
              {mk-rec
               {lambda {[fib : (num -> num)]}
                    {lambda {[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 encodings fail, extend the language and type system

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

```
{letrec {[fib : (num -> num)
                {lambda {[n : num]}
                  {if0 n
                       1
                       {if0 {- n 1}
                            1
                            {+ {fib {- n 1}}
                               {fib {- n 2}}}}}}]}
  {fib 4}}
```

Grammar with Recursion

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  <Sym>
          |  {lambda {[<Sym> : <Type>]} <Expr>}
          |  {<Expr> <Expr>}
          |  {letrec {[<Sym> : <Type> <Expr>]} <Expr>}

<Type>  ::=  num
          |  bool
          |  (<Type> -> <Type>)
```

Datatypes

```
(define-type ExprC
  ...
  [letrecC (name : symbol)
           (rhs-type : Type)
           (rhs : ExprC)
           (body : ExprC)])

(define-type Binding
  [bind (name : symbol)
        (val : (boxof Value))])
```

Interpreter with Recursion

```
(define (interp a env)
  (type-case ExprC env
    ...
    [letrecC (n t rhs body)
             (let ([b (box (numV 42))])
               (let ([new-env (extend-env
                               (bind n b)
                               env)])
                 (begin
                   (set-box! b (interp rhs new-env))
                   (interp body new-env))))]))
```

Type Checking Recursion

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

Abbreviation: **x** = **<Sym>**

Type Checker with Recursion

```
(define typecheck : (ExprC TypeEnv -> Type)
  (lambda (a env)
    (type-case ExprC a
      ...
      [letrecC (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

```
(define typecheck : (ExprC TypeEnv -> Type)
  (lambda (a env)
    (type-case ExprC a
      ...
      [letrecC (n rhs-type rhs body)
               ....
               (typecheck rhs ....)
               (typecheck body ....)
               ....])))
```

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

Type Checker with Recursion

```
(define typecheck : (ExprC TypeEnv -> Type)
  (lambda (a env)
    (type-case ExprC a
      ...
      [letrecC (n rhs-type rhs body)
               (let ([new-tenv (extend-env
                                (tbind n rhs-type)
                                tenv)])
                 ....
                 (typecheck rhs new-tenv)
                 (typecheck body new-tenv)
                 ....)])))
```

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

Type Checker with Recursion

```
(define typecheck : (ExprC TypeEnv -> Type)
  (lambda (a env)
    (type-case ExprC a
      ...
      [letrecC (n rhs-type rhs body)
               (let ([new-tenv (extend-env
                                (tbind n rhs-type)
                                tenv)])
                 (if (equal? rhs-type
                             (typecheck rhs new-tenv))
                     (typecheck body new-tenv)
                     (type-error rhs (to-string rhs-type))))])))
```

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

Part 2

Types of Data

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

Variants

```
(define-type Grade
  [percent (r : number)]
  [pass/fail (pass? : boolean)])

(define-type Chain
  [end (r : number)]
  [link (next : Chain)])

(define-type IntList
  [empty (d : number)]
  [cons (p : (number * IntList))])

...
```

Variants

```
{let {[percent : (num -> (num * (num * bool)))
               {lambda {[x : num]}
                 {pair 0 {pair x false}}}]}
  {let {[passfail : (num -> (num * (num * bool)))
                  {lambda {[y : bool]}
                    {pair 1 {pair 0 y}}}]}
    {let {[pass? : ((num * (num * bool)) -> bool)
                 {lambda {[p : (num * (num * bool))]}
                   {if0 {fst p}
                        {> {fst {snd p}} 70}
                        {snd {snd p}}}}]}
      {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 : (num -> …)
            {lambda {[d : num]}
              {pair 0 d}}]}
  {let {[link : (… -> …)
              {lambda {[r : …]}
                {pair 1 r}}]}
    {link {link {link {end 0}}}}}}
```

Recursive Datatypes

```
{let {[end : (num -> (num * num))
            {lambda {[d : num]}
              {pair 0 d}}]}
  {let {[link : (… -> …)
              {lambda {[r : …]}
                {pair 1 r}}]}
    {link {link {link {end 0}}}}}}
```

Recursive Datatypes

```
{let {[end : (num -> (num * num))
            {lambda {[d : num]}
              {pair 0 d}}]}
  {let {[link : ((num * num) -> …)
              {lambda {[r : …]}
                {pair 1 r}}]}
    {link {link {link {end 0}}}}}}
```

Recursive Datatypes

```
{let {[end : (num -> (num * num))
            {lambda {[d : num]}
              {pair 0 d}}]}
  {let {[link : ((num * num) -> …)
              {lambda {[r : (num * num)]}
                {pair 1 r}}]}
    {link {link {link {end 0}}}}}}
```

Recursive Datatypes

```
{let {[end : (num -> (num * num))
            {lambda {[d : num]}
              {pair 0 d}}]}
  {let {[link : ((num * num) -> (num * (num * num)))
                {lambda {[r : (num * num)]}
                  {pair 1 r}}]}
    {link {link {link {end 0}}}}}}
```

Stuck...

Recursive Datatypes

```
{let {[empty : (num -> (num * …))
              {lambda {[d : num]}
                {pair 0 d}}]}
  {let {[cons : (num -> ((num * …) -> (num * …)))
              {lambda {[x : num]}
                {lambda {[r : (num * …)]}
                  {pair 1 {pair x r}}}}}]}
    {{cons 1} {{cons 2} {{cons 3} {empty 0}}}}}}
```

Stuck again with …

Recursive Datatypes

Add **`let-type`** and **`type-case`**:

```
{let-type {Numlist [empty num]
                   [cons (num * Numlist)]}
  {letrec {[len : (Numlist -> num)
                {lambda {[l : Numlist]}
                  {type-case Numlist l
                    [empty {n} 0]
                    [cons {fxr} {+ 1 {len {snd fxr}}}]}}}]}
    {len {cons {pair 1 {cons {pair 2 {empty 0}}}}}}}}
```

Grammar with Variants

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {- <Expr> <Expr>}
          |  <Sym>
          |  {lambda {[<Sym> : <Type>]} <Expr>}
          |  {<Expr> <Expr>}
          |  {letrec {[<Sym> : <Type> <Expr>]} <Expr>}
          |  {let-type {<Sym> [<Sym> <Type>]                    NEW
                              [<Sym> <Type>]}
               <Expr>}
          |  {type-case <Sym> <Expr>                            NEW
               [<Sym> {<Sym>} <Expr>]
               [<Sym> {<Sym>} <Expr>]}

<Type>  ::=  num
          |  bool
          |  (<Type> -> <Type>)
          |  <Sym>                                              NEW
```

Part 3

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                       ...})
              mt-env)
      ...)
```

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                        1})
              mt-env)
      (numV 1))
```

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                       {apple 1}})
              mt-env)
      ....)
```

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                       {apple 1}})
              mt-env)
      (appleV (numV 1))) ; ???
```

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                       {apple 1}})
              mt-env)
      (variantV 'apple (numV 1)))
```

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                       {banana {lambda {[x : num]} 5}}})
              mt-env)
      (variantV 'banana (closV ....)))
```

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                        apple})
              mt-env)
      ....)
```

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                        apple})
              mt-env)
      (constructorV 'apple))
```

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                        banana})
              mt-env)
      (constructorV 'banana))
```

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                       {type-case Fruit {apple 3}
                         [apple {a} a]
                         [banana {b} {b 7}]}})
              mt-env)
      (numV 3))
```

Examples

```
(test (interp (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                        {type-case Fruit {banana
                                          {lambda {[x : num]}
                                            {+ x 3}}}
                          [apple {a} a]
                          [banana {b} {b 7}]}})
              mt-env)
      (numV 10))
```

Part 4

Value Datatype

```
(define-type Value
  ....
  [variantV (tag : symbol)
            (val : Value)]
  [constructorV (tag : symbol)])
```

Expressions for Variants

```
(define-type ExprC
  ....
  [let-typeC (type-name : symbol)
             (tag1 : symbol)
             (type1 : Type)
             (tag2 : symbol)
             (type2 : Type)
             (body : ExprC)]
  [type-caseC (type-name : symbol)
              (tst : ExprC)
              (tag1 : symbol)
              (n1 : symbol)
              (rhs1 : ExprC)
              (tag2 : symbol)
              (n2 : symbol)
              (rhs2 : ExprC)])

(define-type Type
  ....
  [definedT (name : symbol)])
```

Interpreter

```
(define (interp a tenv)
  (type-case ExprC a
    ....
    [let-typeC (type-name tag1 type1
                          tag2 type2
                          body)
               (interp body
                       (extend-env
                        (bind tag1
                              (constructorV tag1))
                        (extend-env
                         (bind tag2
                               (constructorV tag2))
                         env)))]
    ....))
```

Interpreter

```
(define (interp a tenv)
  (type-case ExprC a
    ....
    [appC (fun arg)
          (type-case Value (interp fun env)
            [closV (n body c-env) ....]
            [constructorV (tag)
                          (variantV tag (interp arg env))]
            [else (error 'interp "not a function")])]
    ....))
```

Interpreter

```
(define (interp a tenv)
  (type-case ExprC a
    ....
    [type-caseC (type-name tst
                           tag1 n1 rhs1
                           tag2 n2 rhs2)
     (type-case Value (interp tst env)
       [variantV (tag val)
                 (cond
                  [(eq? tag tag1)
                   (interp rhs1
                           (extend-env
                            (bind n1 (box val))
                            env))]
                  [(eq? tag tag2)
                   (interp rhs2
                           (extend-env
                            (bind n2 (box val))
                            env))]
                  [else (error 'interp "wrong tag")])]
       [else (error 'interp "not a variant")])]
    ....))
```

Part 5

Typechecking Examples

```
(test (typecheck
       (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                 ...})
       mt-env)
      ...)
```

Typechecking Examples

```
(test (typecheck
       (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                 1})
       mt-env)
      (numT))
```

Typechecking Examples

```
(test (typecheck
       (parse '{let-type {Fruit [apple num]
                                [banana (num -> num)]}
                 {type-case Fruit {apple 3}
                   [apple {a} a]
                   [banana {b} {b 7}]}})
       mt-env)
      (numT))
```

Typechecking Examples

```
(test/exn (typecheck
           (parse '{let-type {Fruit [apple num]
                                    [banana (num -> num)]}
                     {type-case Fruit {apple 3}
                       [radish {a} a]
                       [banana {b} {b 7}]}})
           mt-env)
          "no type")
```

typecheck must record variant names for **let-type**

Typechecking Examples

```
(test/exn (typecheck
           (parse '{let-type {Fruit [apple num]
                                    [banana (num -> num)]}
                     {type-case Fruit {apple 3}
                       [apple {a} a]
                       [banana {b} b]}})
           mt-env)
          "no type")
```

`typecheck` must record variant types for **`let-type`**

Part 6

Bindings for Defined Types

```
{let-type {D [x1 τ1]
             [x2 τ2]}
  ...}
```

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

Bindings for Defined Types

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

```
(define-type TypeBinding
  ....
  [tdef (type-name : symbol)
        (tag1 : symbol)
        (type1 : Type)
        (tag2 : symbol)
        (type2 : Type)])
```

Local Type Names

- Might be ok:

```
{let-type {Fruit [apple num]
                 [banana (num -> num)]}
          ... {lambda {[x : Fruit]} ...} ...}
```

- Not ok:

```
{lambda {[x : Fruit]} ...}
```

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

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

Local Type Names

- Might be ok:

```
{let-type {Fruit [apple num]
                 [banana (num -> num)]}
          ... {lambda {[x : Fruit]} ...} ...}
```

- Not ok:

```
{lambda {[x : Fruit]} ...}
```

tvarcheck

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

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

Type Checking

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma[\texttt{<Sym>} \leftarrow \tau_1] \vdash \texttt{e} : \tau_2}{\Gamma \vdash \{\texttt{lambda}\ \{[\texttt{<Sym>} : \tau_1]\}\ \texttt{e}\} : (\tau_1 \rightarrow \tau_2)}$$

```
(define (typecheck a tenv)
  (type-case ExprC a
    ....
    [lamC (n arg-type body)
          (begin
            (tvarcheck arg-type tenv)
            (arrowT arg-type
                    (typecheck body
                               (extend-env (tbind n arg-type)
                                           tenv))))]
    ....))
```

Type Checker with Variants

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

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

Warning: this `let-type` rule is not quite right...

Type Checking

```
(define (typecheck a tenv)
  (type-case ExprC a
    ....
    [let-typeC (type-name tag1 type1
                          tag2 type2
                          body)
     (let ([new-tenv (extend-env (tdef type-name
                                       tag1 type1
                                       tag2 type2)
                                 tenv)])
       (begin
         (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

```
(define (typecheck a tenv)
  (type-case ExprC a
    ....
    [type-caseC (type-name tst tag1 n1 rhs1
                           tag2 n2 rhs2)
     (let ([def (defined-type-lookup type-name tenv)])
       ....
       (type-case Type (typecheck tst tenv)
         [definedT (name)
           .... (equal? name type-name)
           .... (typecheck rhs1
                           (extend-env
                            (tbind n1 (tdef-type1 def))
                            tenv))
           .... (typecheck rhs2
                           (extend-env
                            (tbind n2 (tdef-type2 def))
                            tenv))
           ....])
       ....)]
    ....))
```

Type Checking

```
(define (typecheck a tenv)
  (type-case ExprC a
    ....
    [type-caseC (type-name tst tag1 n1 rhs1
                           tag2 n2 rhs2)
     (let ([def (defined-type-lookup type-name tenv)])
       (if (not (and (equal? tag1 (tdef-tag1 def))
                     (equal? tag2 (tdef-tag2 def))))
           (type-error a "matching variant names")
           (type-case Type (typecheck tst tenv)
             [definedT (name)
               (if (not (equal? name type-name))
                   (type-error tst (to-string type-name))
                   (let ([rhs1-t (typecheck rhs1
                                            (extend-env
                                             (tbind n1 (tdef-type1 def))
                                             tenv))]
                         [rhs2-t  (typecheck rhs2
                                             (extend-env
                                              (tbind n2 (tdef-type2 def))
                                              tenv))])
                     (if (equal? rhs1-t rhs2-t)
                         rhs1-t
                         (type-error rhs2 (to-string rhs1-t)))))]
             [else (type-error tst (to-string type-name))])))]
    ....))
```

Checking Type Forms

```
(define (tvarcheck ty tenv)
  (type-case Type ty
    [numT () (values)]
    [boolT () (values)]
    [arrowT (a b) (begin
                    (tvarcheck a tenv)
                    (tvarcheck b tenv))]
    [definedT (id) (begin
                     (defined-type-lookup id tenv)
                     (values))]))
```

Part 7

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 : (num -> num)
          f]}
  {f 10}}
```

Solution 1: change the grammar for **`letrec`**

```
<Expr> ::= ...
         | {letrec {[<Sym> : <Type>
                     {lambda {[<Sym> : <Type>]}
                       <Expr>}]}
             <Expr>}
```

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

... best with an "undefined" value

Type Soundness

`let-type` checking has a bug, too:

```
{{let-type {Foo [a num] [b num]}
   {lambda {x : Foo} {+ 1 {type-case Foo x
                            [a {n} n]
                            [b {n} n]}}}}
 {let-type {Foo [a (num -> num)] [b num]}
   {a {lambda {[y : num]} y}}}}
```

Solution 1: no local type declarations

Solution 2: don't let **D** escape **`let-type`**

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