System F_ω with Equirecursive Types for Datatype-Generic Programming

Yufei Cai, Paolo G. Giarrusso, Klaus Ostermann
University of Tübingen, Germany

- There are many useful ways to **view** a datatype.
- Often in datatype-generic programming (DGP), one single **view** is locked in. **Tyranny of the Dominant Functor**.
- We designed a calculus to support DGP free from Tyranny.

Cover art of *Gödel, Escher, Bach: An Eternal Golden Braid* by Douglas Hofstadter

System F_ω with Equirecursive Types for Datatype-Generic Programming

3. Solution
4. Consequences

System F_ω with Equirecursive Types for Datatype-Generic Programming

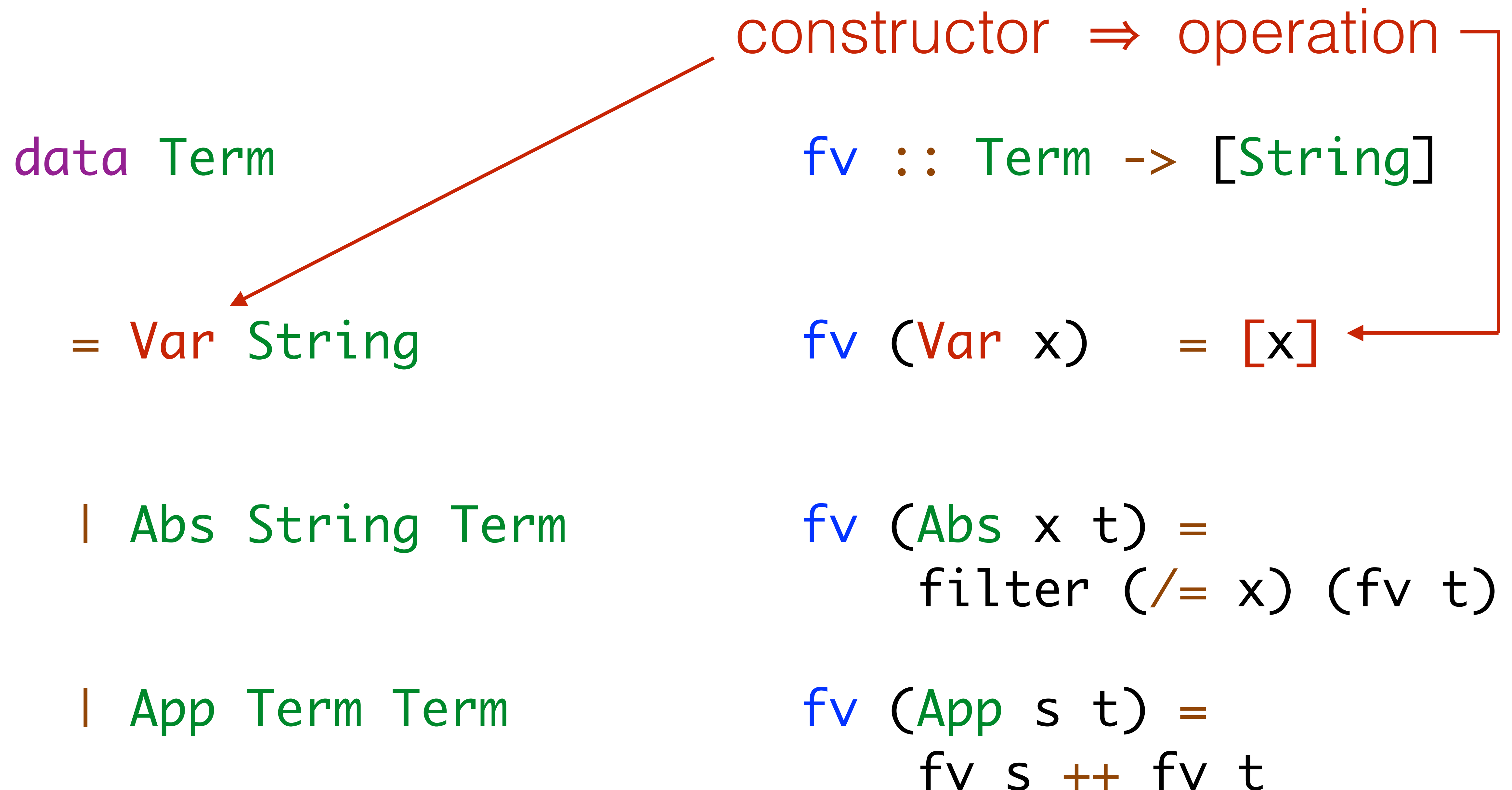
1. Background
2. Problem statement

Background

```haskell
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```haskell
fv :: Term -> [String]
fv (Var x)    = [x]
fv (Abs x t) =
    filter (/= x) (fv t)
fv (App s t) =
    fv s ++ fv t
```

Background



Background

```
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```
fv :: Term -> [String]
fv (Var x)   = [x]
fv (Abs x t) =
    filter (/= x) (fv t)
fv (App s t) =
    fv s ++ fv t
```

Background

```
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```
fv :: Term -> [String]
fv (Var x)    = [x]
fv (Abs x t) =
      filter (/= x) (fv t)
fv (App s t) =
      fv s ++ fv t
```

Background

```
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```
fv :: Term -> [String]
fv (Var x)   = [x]
fv (Abs x t) =
    filter (/= x) (fv t)
fv (App s t) =
    fv s ++ fv t
```

type recursion ⇒ recursive calls

Background

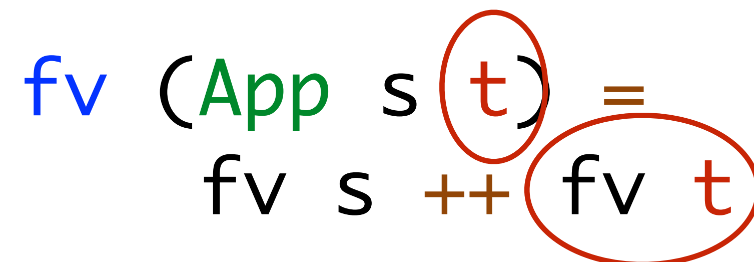
```
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```
fv :: Term -> [String]
fv (Var x)   = [x]
fv (Abs x t) =
    filter (/= x) (fv t)
fv (App s t) =
    fv s ++ fv t
```

Background

```haskell
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```haskell
fv :: Term -> [String]
fv (Var x)    = [x]
fv (Abs x t) =
    filter (/= x) (fv t)
fv (App s t) =
    fv s ++ fv t
```



Background

```
type Term1 = Fix TermF

data TermF a
  = Var String
  | Abs String a
  | App a a
```

recursive calls

```
data Fix f
  = In (f (Fix f))

fold1 :: (TermF a -> a)
      -> Term1 -> a

fv = fold1 fvAlgebra
```

Background

```
type Term₁ = Fix TermF

data TermF a
  = Var String
  | Abs String a
  | App a a
```

recursive calls

```
data Fix f
  = In (f (Fix f))
```

```
fold₁ :: (TermF a -> a)
      -> Term₁ -> a

fv = fold₁ fvAlgebra
```

operations for constructors

Problem

```haskell
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```haskell
args :: Term -> [Term]
args (Var x)   = []
args (Abs x t) = []
args (App s t) =
    args s ++ [t]
```

Problem

```
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```
args :: Term -> [Term]
args (Var x)   = []
args (Abs x t) = []
args (App s t) =
  args s ++ [t]
```

recursive call

Problem

```
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```
args :: Term -> [Term]
args (Var x)   = []
args (Abs x t) = []
args (App s t) =
     args s ++ [t]
```

no recursive call!

Problem

```
type Term₂ = Fix ArgsF

data ArgsF a
  = Var String
  | Abs String Term
  | App a Term
```

recursive calls

```
data Fix f
  = In (f (Fix f))

fold₂ :: (ArgsF a -> a)
      -> Term₂ -> a

args = fold₂ argsAlgebra
```

Problem

```
compile code =
  let
    t  = parse code :: Term
    …
    vs = fv t
    …
    xs = args t
    …
  in
    …
```

Problem

```
Term ≅ Fix TermF
     ≅ Fix ArgsF
     ≅ Fix (TermF ∘ TermF)
     ≅ ArgsF (ArgsF (Fix (TermF ∘ TermF ∘ ArgF)))
```

Problem

- Either `fv` or `args` can be written in terms of `fold` but not both
- Which of them can be refactored depends on the representation of `Term`
- Tyranny of the dominant functor

Solution idea

```
Term ≅ Fix TermF

     ≅ Fix ArgsF

     ≅ Fix (TermF ∘ TermF)

     ≅ ArgsF (ArgsF (Fix (TermF ∘ TermF ∘ ArgF)))
```

Solution idea

```
Term = Fix TermF
     = Fix ArgsF
     = Fix (TermF ∘ TermF)
     = ArgsF (ArgsF (Fix (TermF ∘ TermF ∘ ArgF)))
```

Solution: F^{μ}_{ω}

```
data Term                          type Term
                                     = μ (λ τ.
  = Var String                         < var : String
  | Abs String Term                    , abs : {x : String,
                                                body : τ  }
  | App Term Term                      , app : {fun : τ,
                                                arg : τ}
                                       >)
```

Features:

Solution: F^{μ}_{ω}

```
data Term                       type Term
                                  = μ (λ τ.
  = Var String                      < var : String
  | Abs String Term                 , abs : {x : String,
                                             body : τ  }
  | App Term Term                   , app : {fun : τ,
                                             arg : τ}
                                    >)
```

Features: Type **recursion**, type **function**

Solution: F^{μ}_{ω}

```
data Term                              type Term
                                         = μ (λ τ.
  = Var String                             < var : String
  | Abs String Term                        , abs : {x : String,
                                                    body : τ  }
  | App Term Term                          , app : {fun : τ,
                                                    arg : τ}
                                           >)
```

Features: Type recursion, type function, **record**

Solution: F_ω^μ

```
data Term                          type Term
                                     = μ (λ τ.

  = Var String                         < var : String


  | Abs String Term                    , abs : {x : String,
                                                body : τ  }

  | App Term Term                      , app : {fun : τ,
                                                arg : τ}

                                       >)
```

Features: Type recursion, type function, **record**

Solution: F_ω^μ

```
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```
type Term
  = μ (λ τ.
    < var : String
    , abs : {x : String,
             body : τ  }
    , app : {fun : τ,
             arg : τ}
    >)
```

Features: Type recursion, type function, record, **variant**

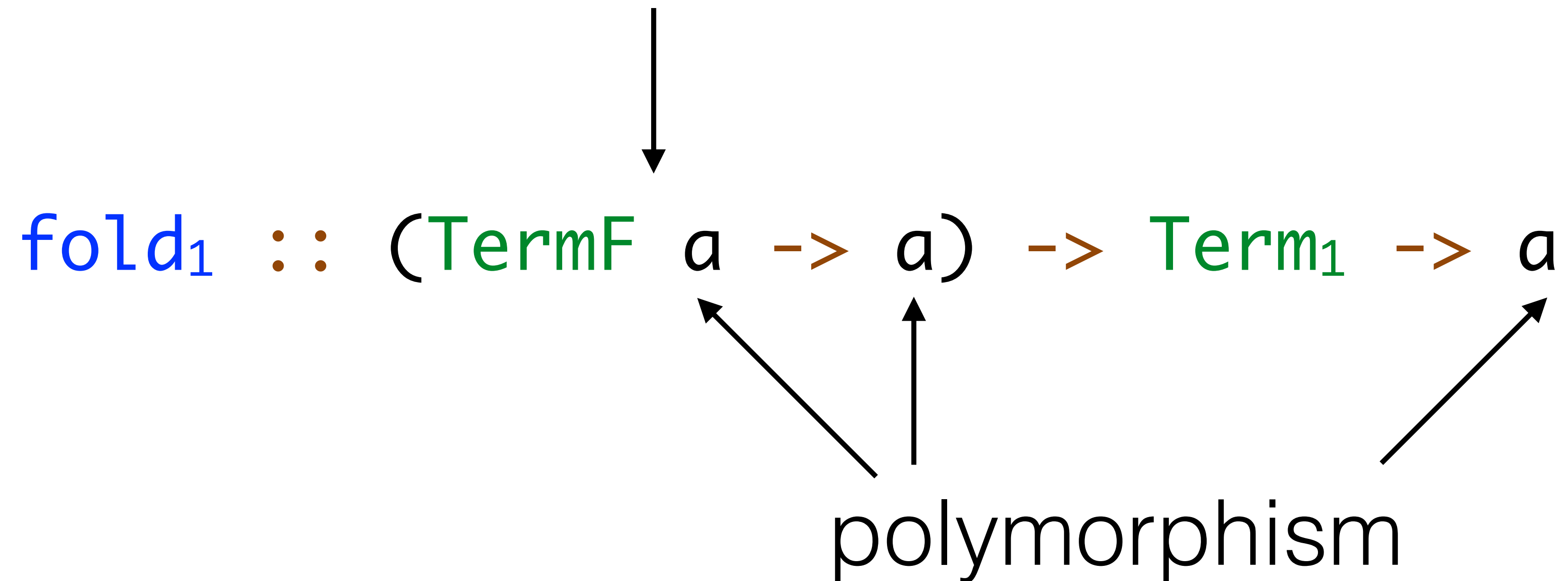
Solution: F_ω^μ

```
data Term
  = Var String
  | Abs String Term
  | App Term Term
```

```
type Term
  = μ (λ τ.
      < var : String
      , abs : {x : String,
               body : τ  }
      , app : {fun : τ,
               arg : τ}
      >)
```

Features: Type recursion, type function, record, variant

Solution: F_ω^μ



Features: Type recursion, type function, record, variant

Solution: F_ω^μ

- Type function
- Type-level application
- polymorphism
- Type recursion
- Record
- Variant

Solution: $\mu f = f (\mu f)$

```
type Term₁
  = μ (λ a.
    < var : String
    , abs : {x : String,
             body : a   }
    , app : {fun : a,
             arg : a}
    >)
```

```
type Term₂
  = μ (λ a.
    < var : String
    , abs : {x : String,
             body : Term}
    , app : {fun : a,
             arg : Term}
    >)
```

Solution: μ f = f (μ f)

```
type Term₁
  = μ (λ a.
    < var : String
    , abs : {x : String,
             body : a  }
    , app : {fun : a,
             arg : a}
    >)
```

```
type Term₂
  = μ (λ a.
    < var : String
    , abs : {x : String,
             body : Term}
    , app : {fun : a,
             arg : Term}
    >)
```

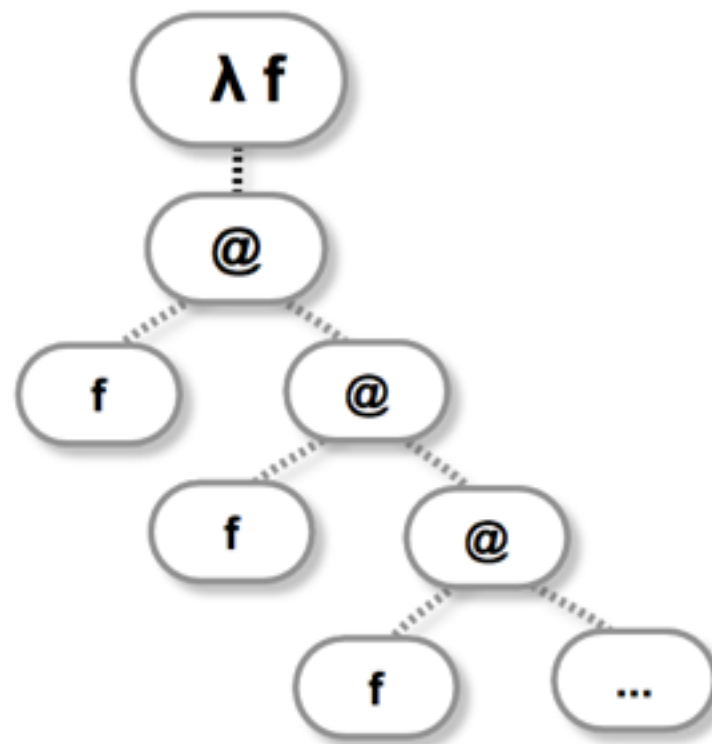
Solution: μ f = f (μ f)

```
type Term₁
  = μ (λ a.
    < var : String
    , abs : {x : String,
             body : a   }
    , app : {fun : a,
             arg : a}
    >)
```

```
type Term₂
  = μ (λ @.
    < var : String
    , abs : {x : String,
             body : Term}
    , app : {fun : @,
             arg : Term}
    >)
```

Solution: $\mu\ f = f\ (\mu\ f)$

μ =

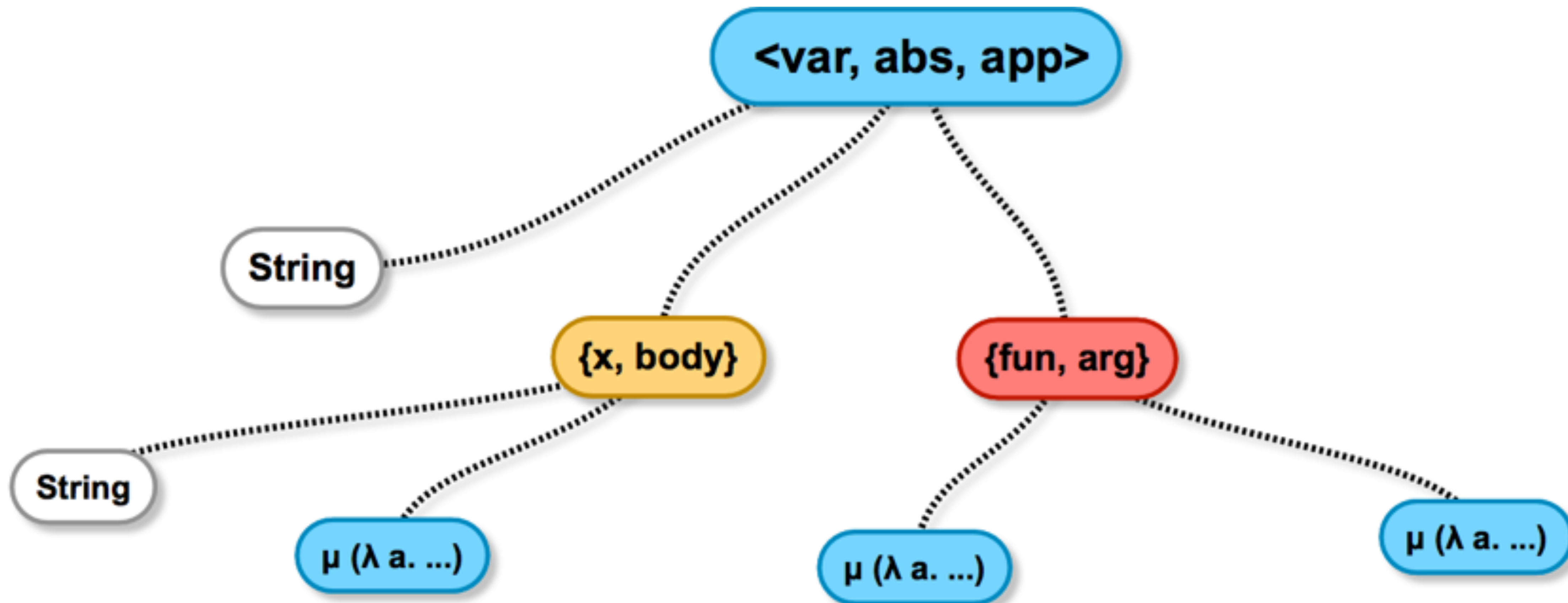


$\mu\ =\ \lambda\ f.\ f\ (f\ (f\ \ldots))$

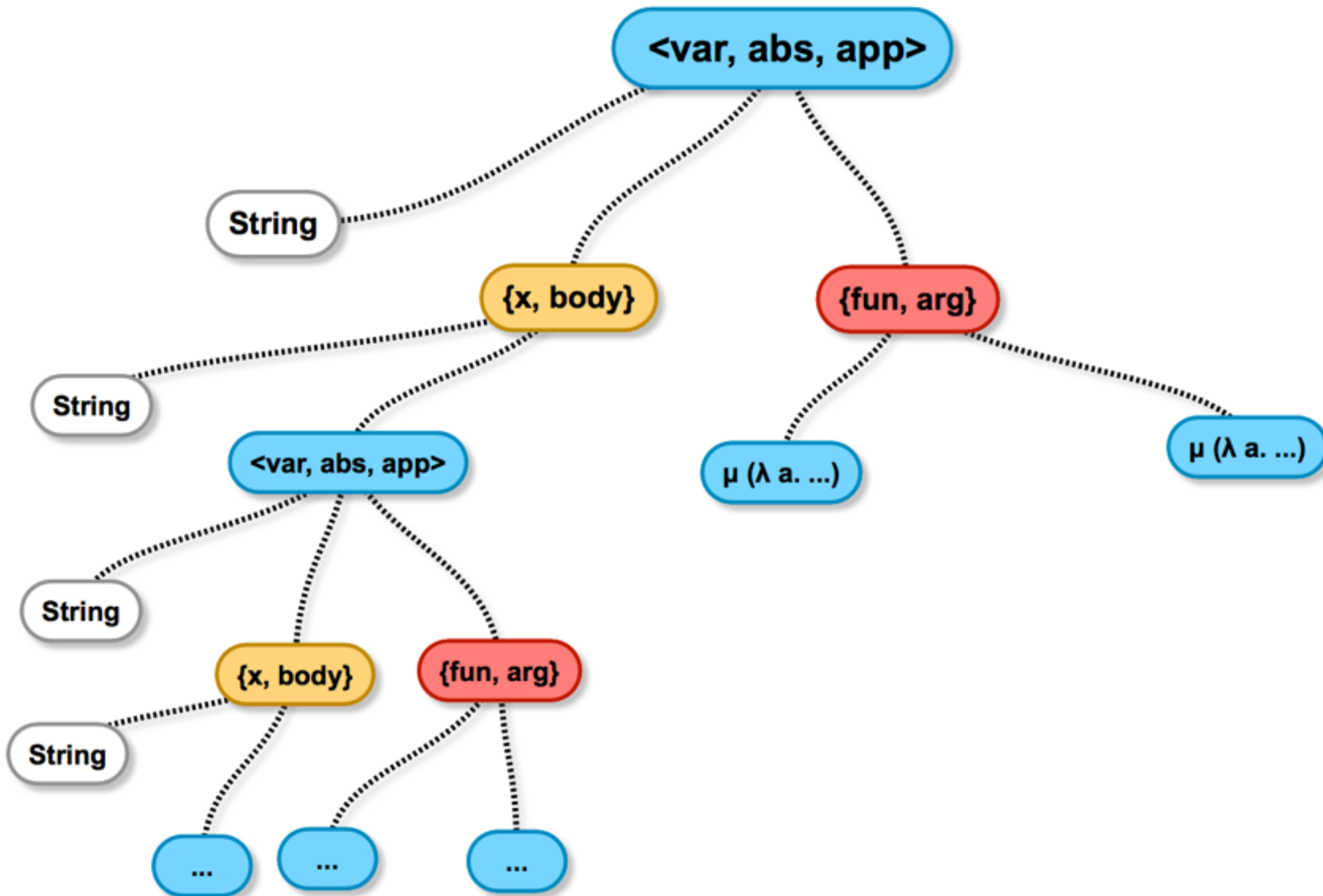
```
Term1 = μ (λ a. < var : String, abs : {x : String, body : a   }
                                , app : {fun : a, arg : a}     >)
```

μ (λ a. ...)

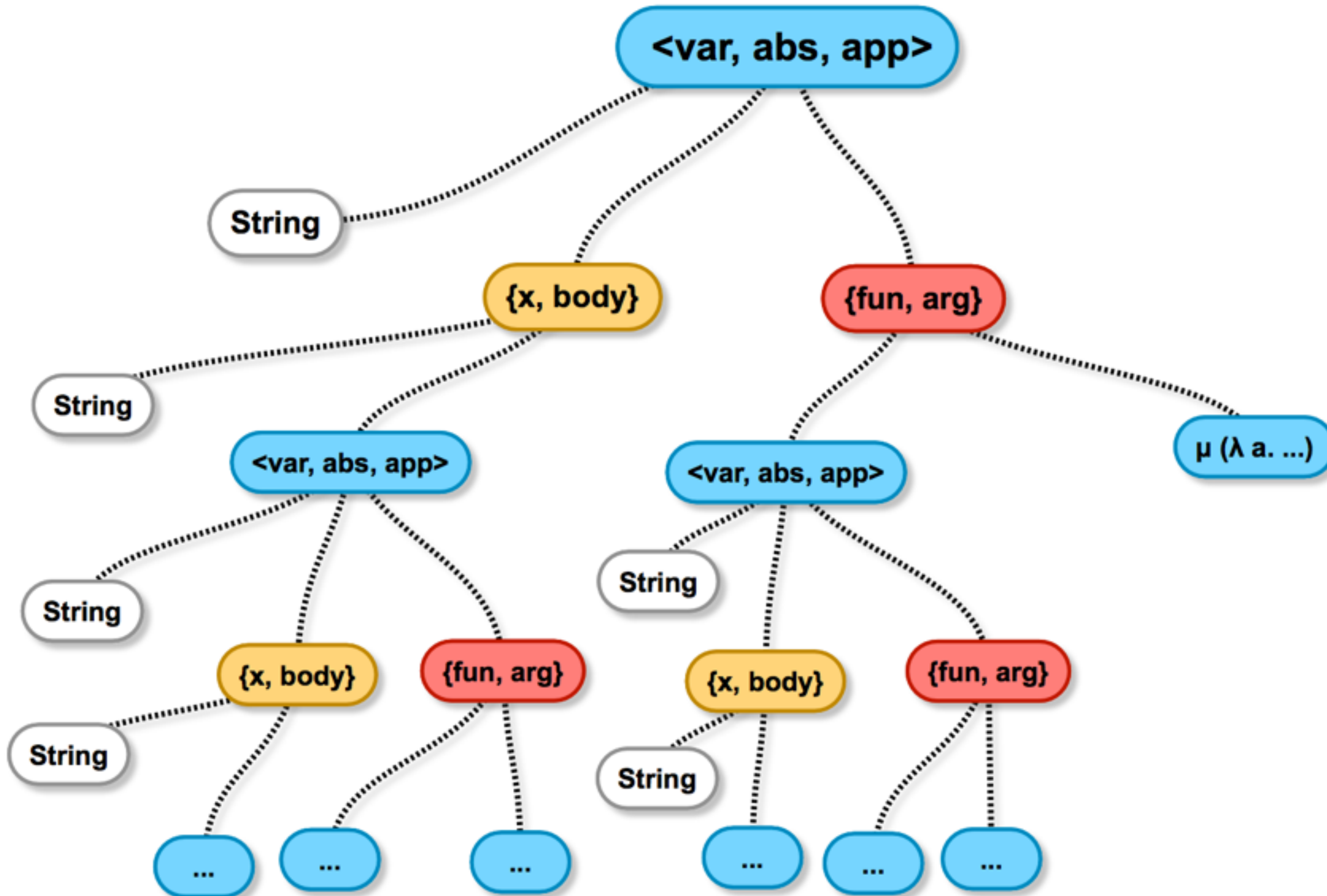
```
Term₁ = μ (λ a. < var : String, abs : {x : String, body : a   }
                                  , app : {fun : a, arg : a}     >)
```



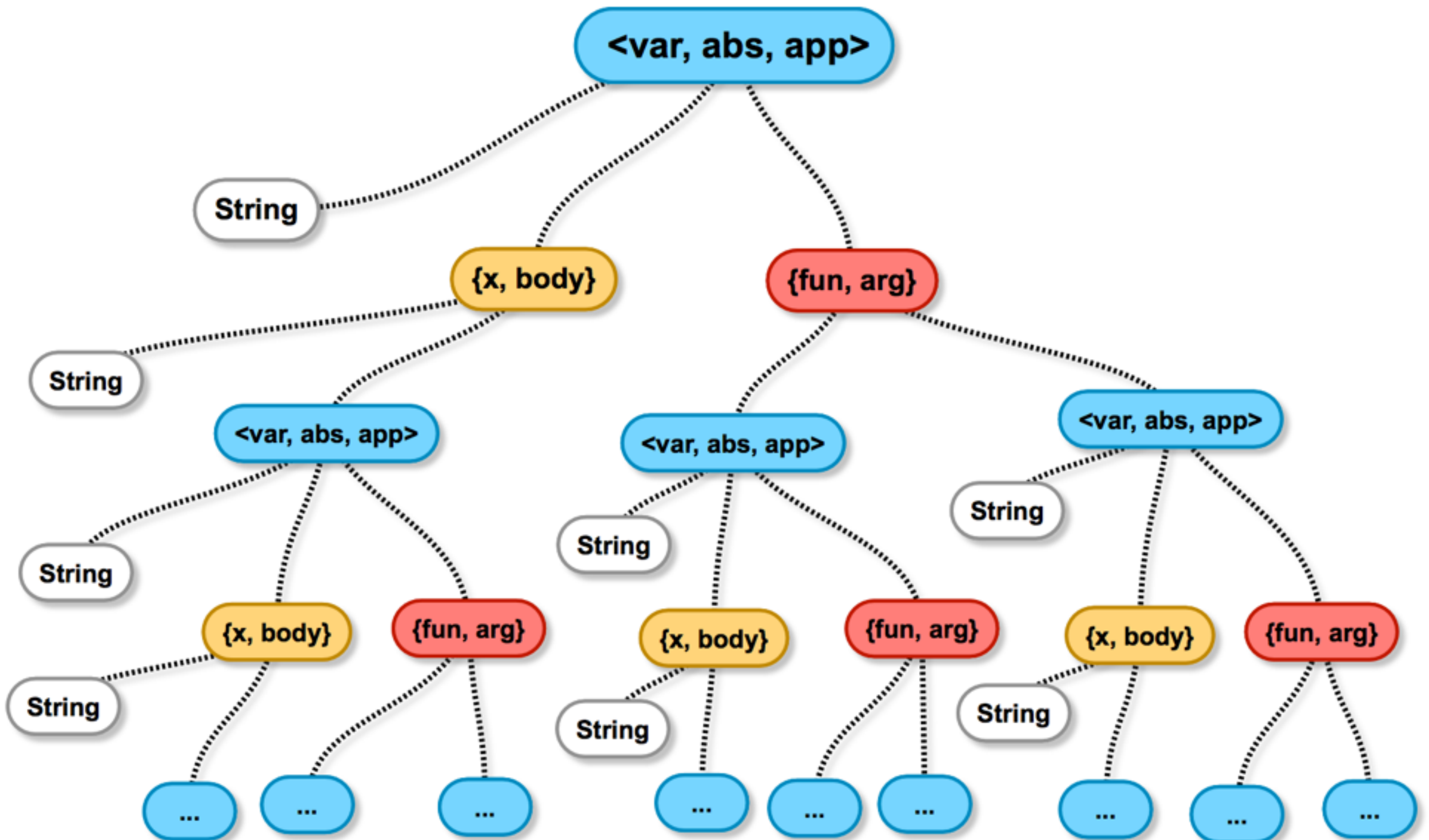
```
Term₁ = μ (λ a. < var : String, abs : {x : String, body : a  }
                                , app : {fun : a, arg : a}     >)
```



```
Term1 = μ (λ a. < var : String, abs : {x : String, body : a  }
                                , app : {fun : a, arg : a}     >)
```



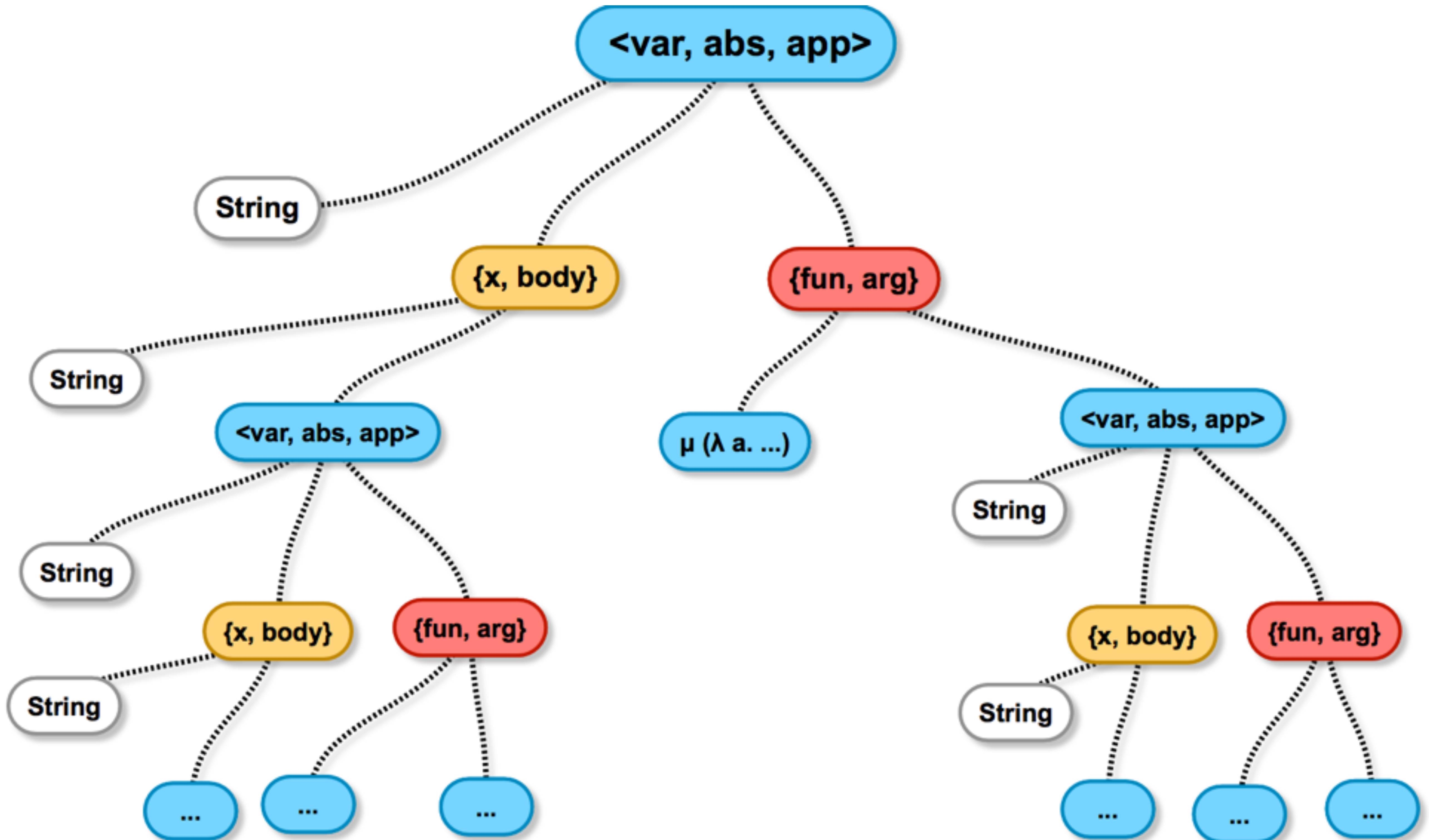
```
Term₁ = μ (λ a. < var : String, abs : {x : String, body : a  }
                                  , app : {fun : a, arg : a}      >)
```



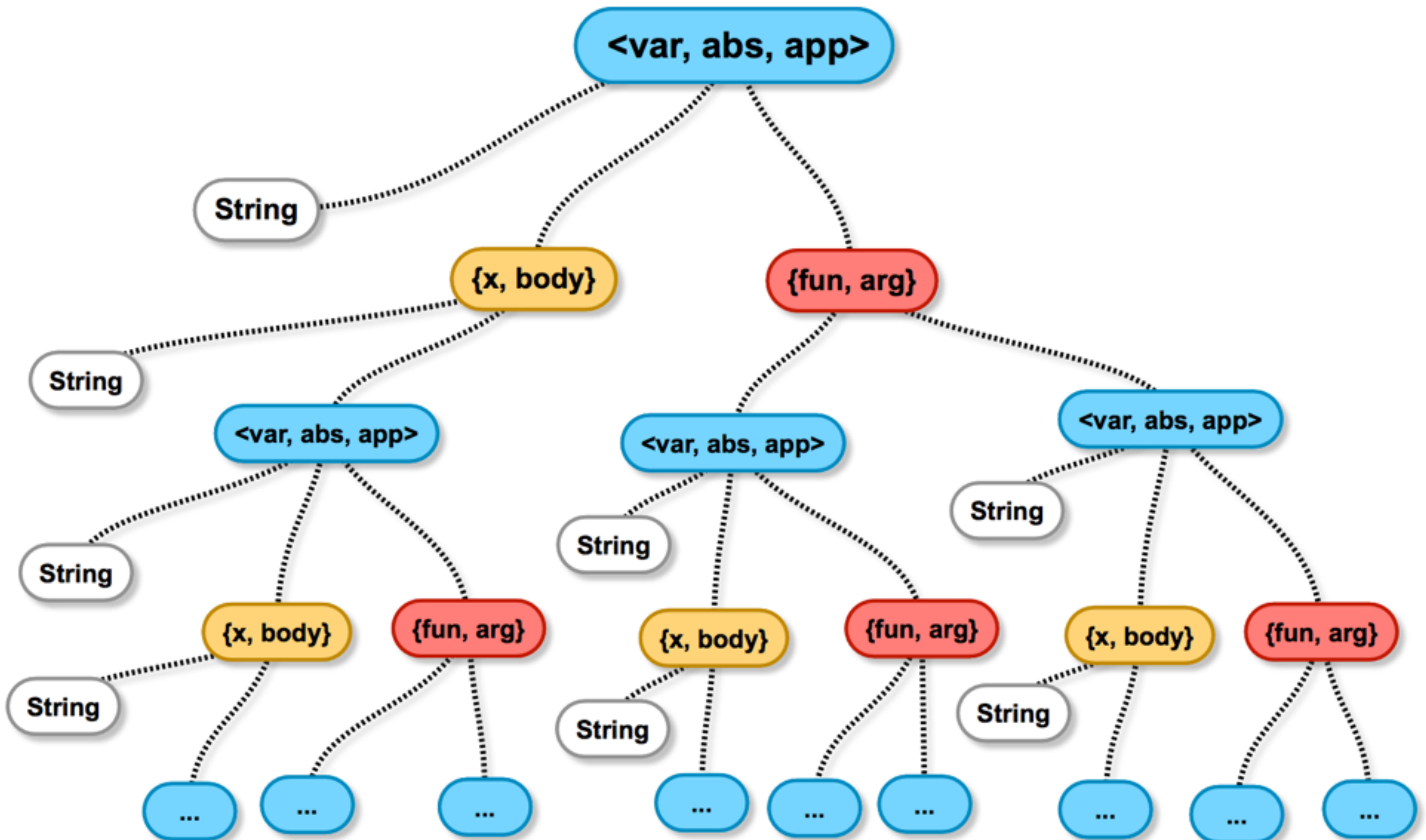
```
Term₂ = μ (λ a. < var : String, abs : {x : String, body : Term }
                                , app : {fun : a, arg : Term}    >)
```

μ (λ a. ...)

```
Term2 = μ (λ a. < var : String, abs : {x : String, body : Term }
                               , app : {fun : a, arg : Term}    >)
```



```
Term2 = μ (λ a. < var : String, abs : {x : String, body : Term }
                , app : {fun : a, arg : Term}    >)
```



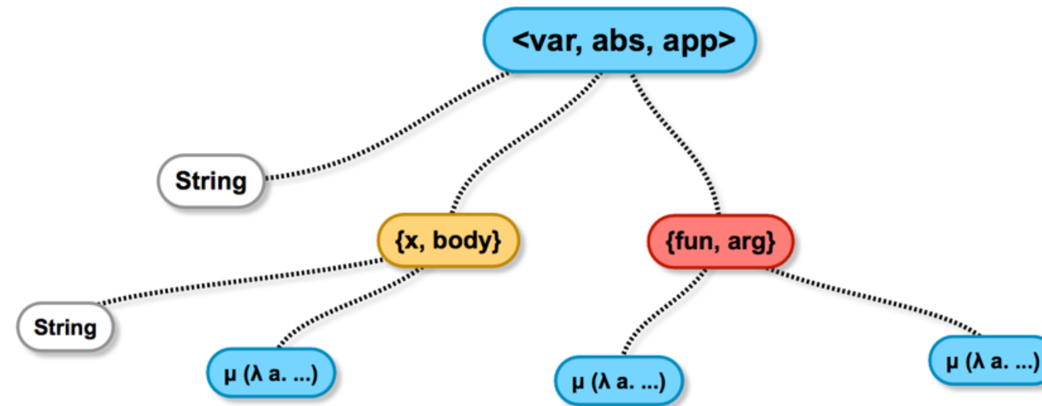
μ (λ a. ...)

$Term_1$ =

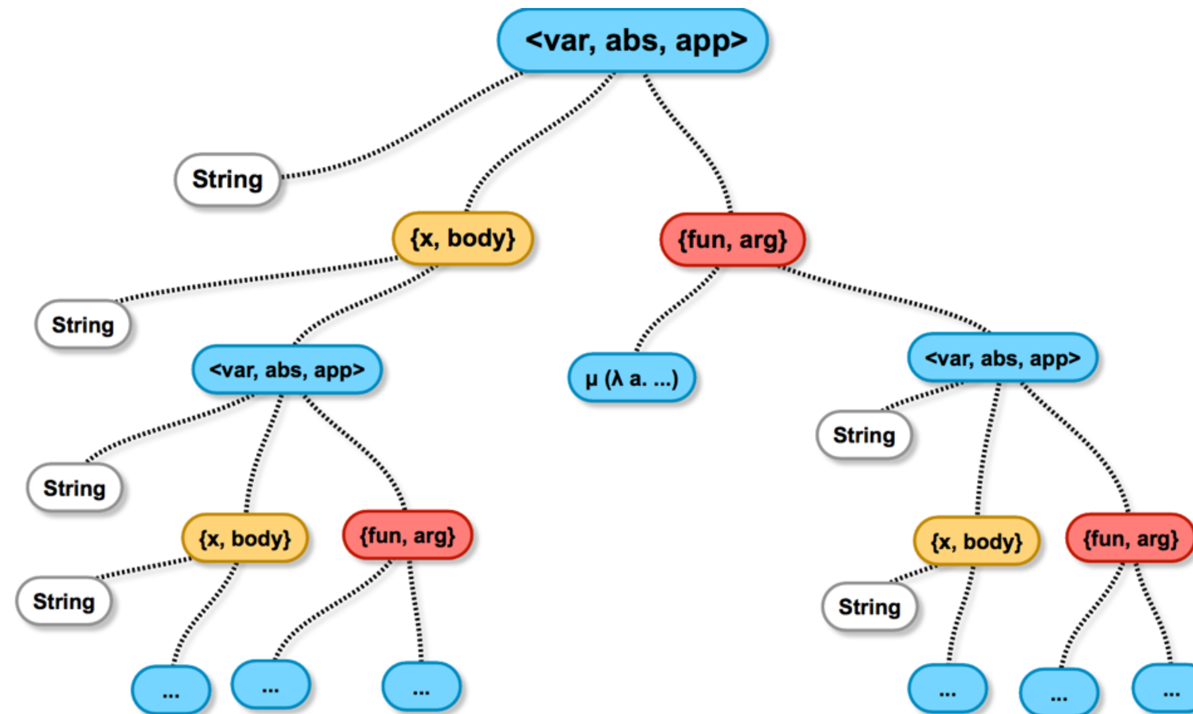
μ (λ a. ...)

$Term_2$ =

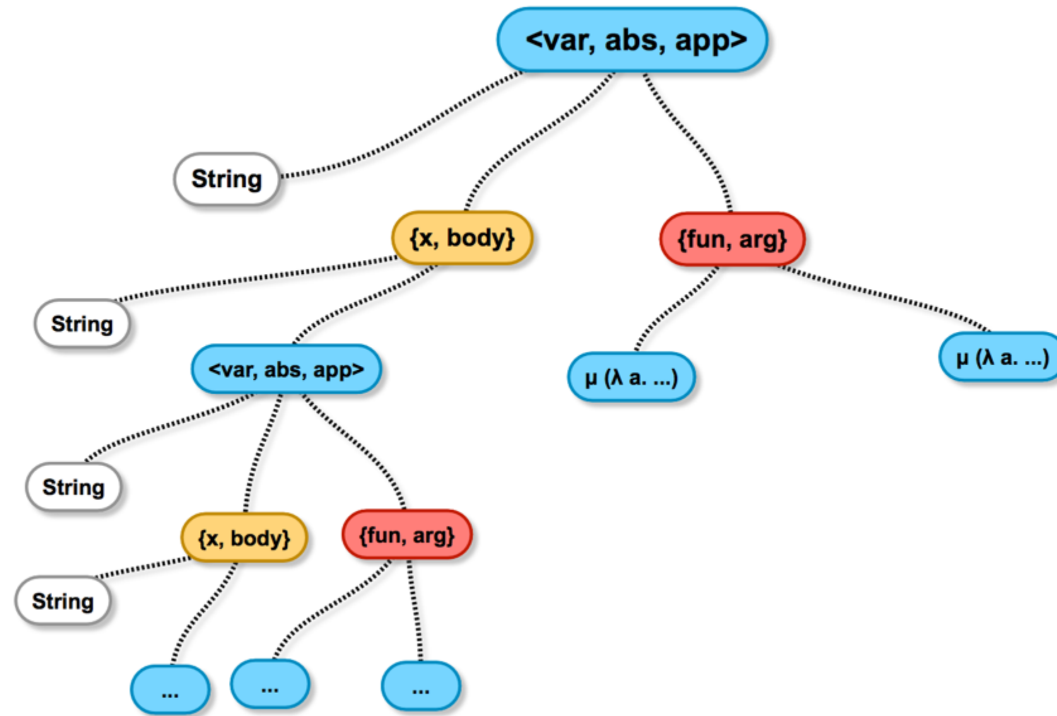
$Term_1 =$



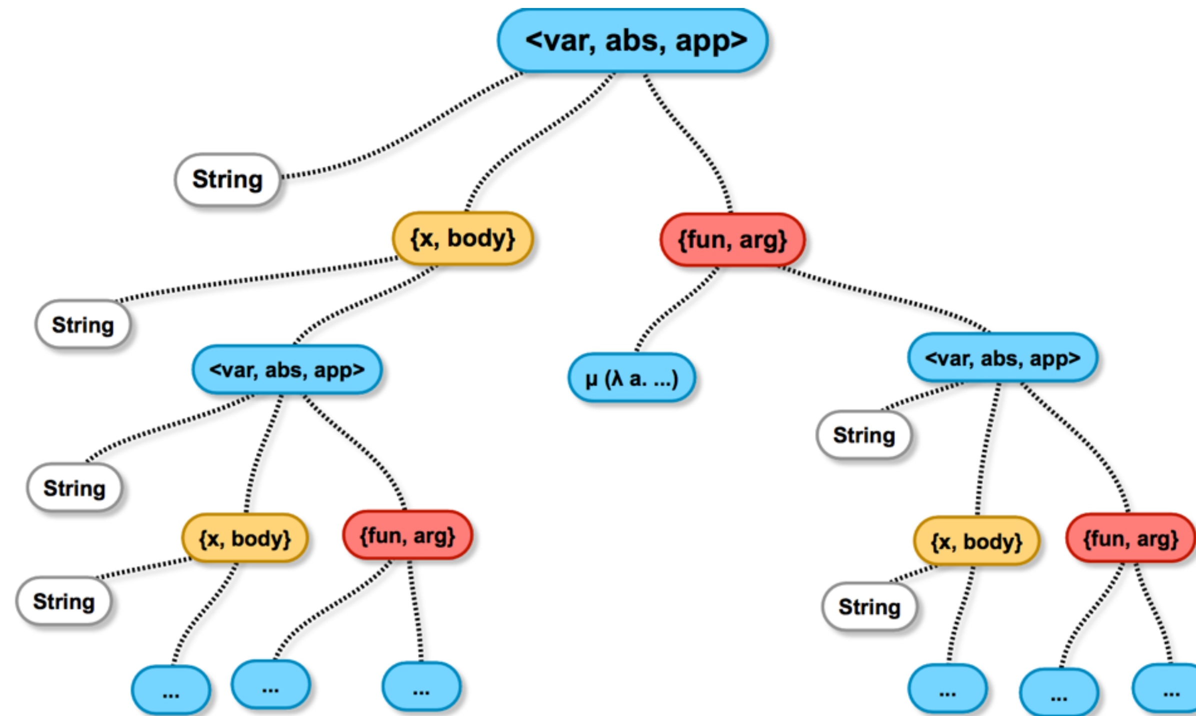
$Term_2 =$



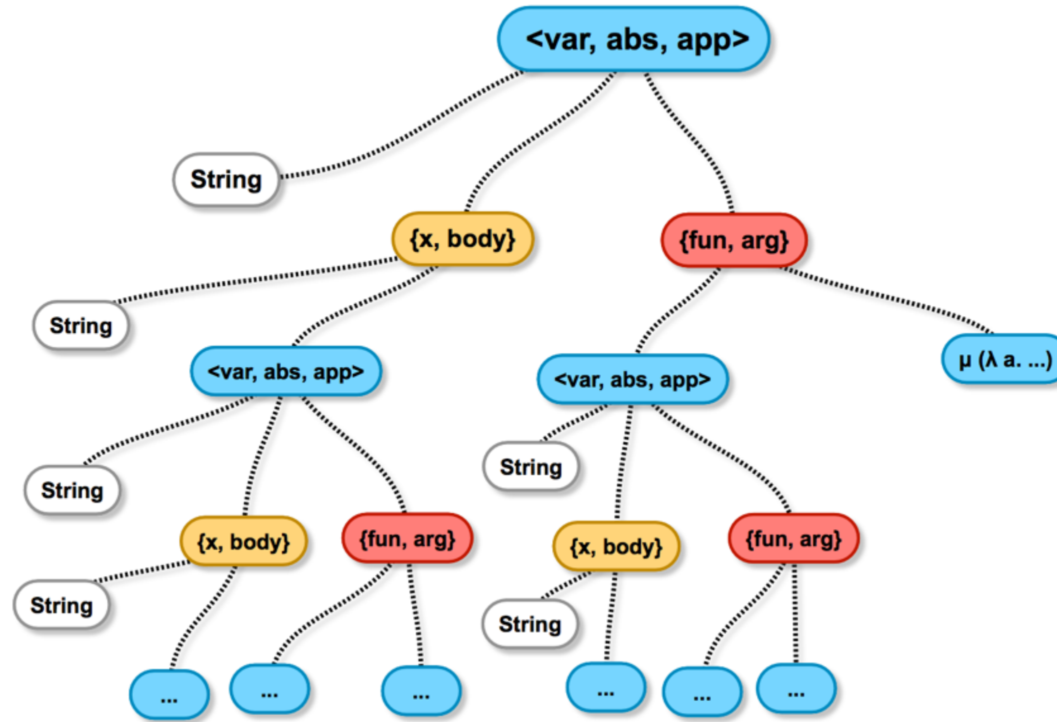
$Term_1$ =



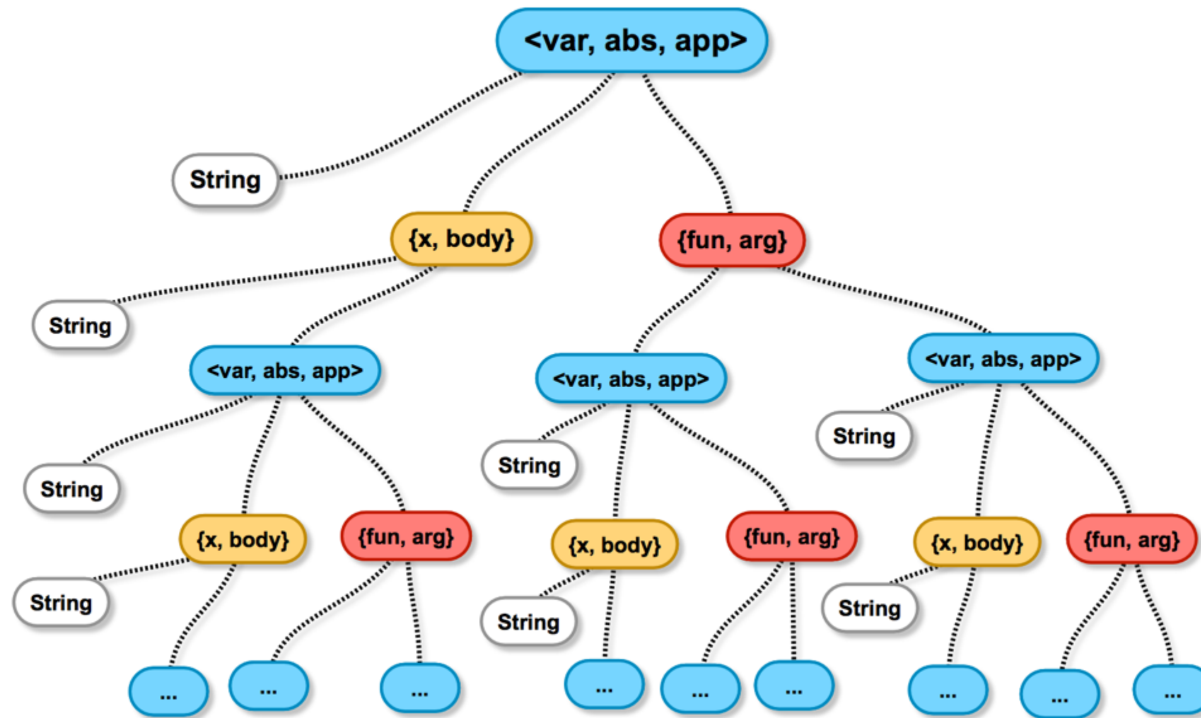
$Term_2$ =



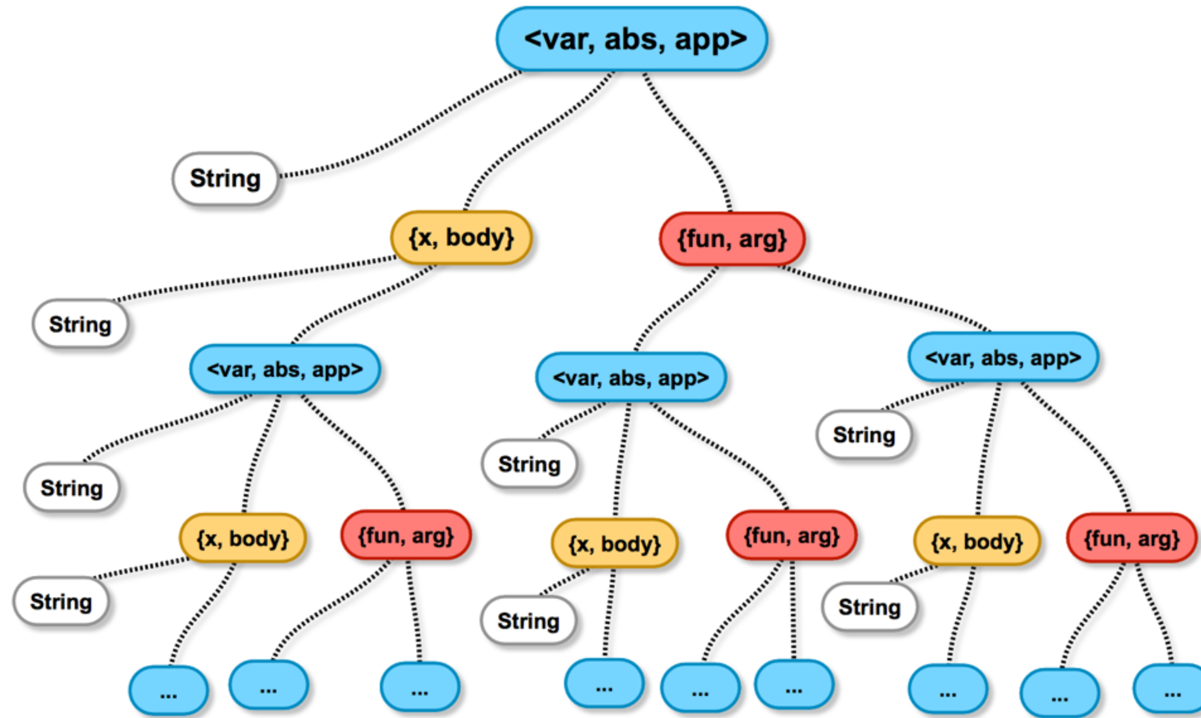
$\mathtt{Term}_1$ =



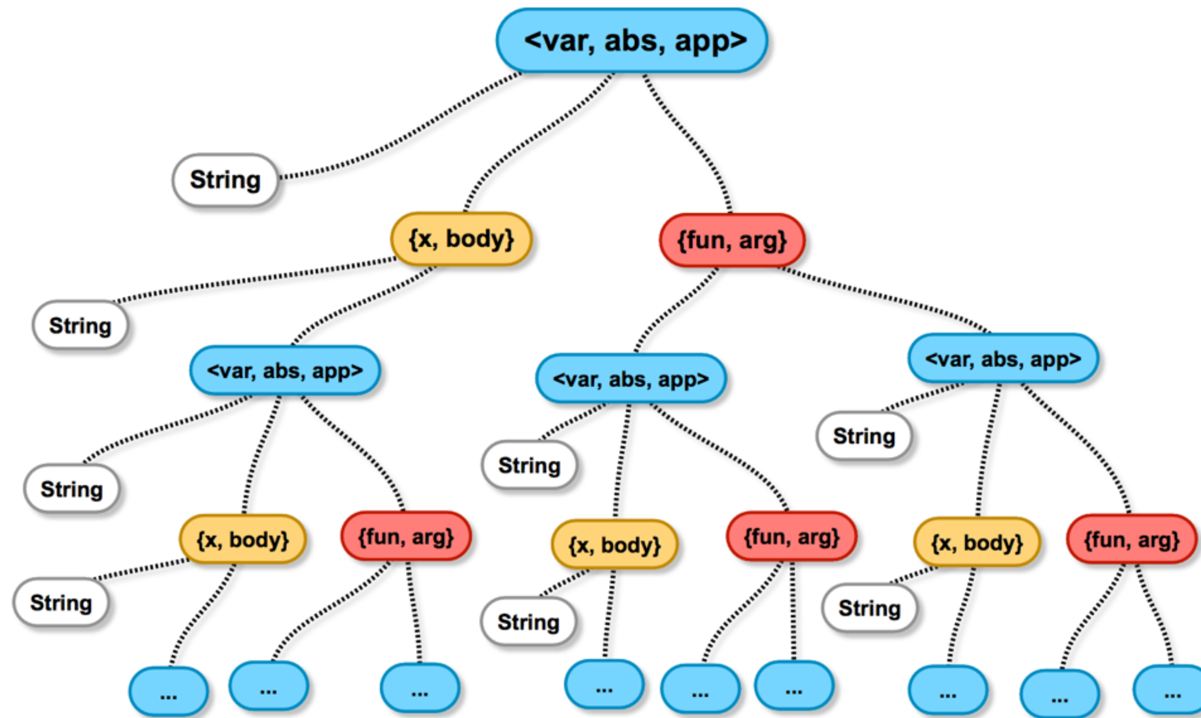
$\mathtt{Term}_2$ =



$\mathtt{Term}_1$ =



$\mathtt{Term}_2$ =



Type equality achieved

```
Term = Fix TermF

     = Fix ArgsF

     = Fix (TermF ∘ TermF)

     = ArgsF (ArgsF (Fix (TermF ∘ TermF ∘ ArgF)))
```

Tyranny ended

```
compile code =
  let
    t  = parse code :: Term
    …
    vs = fv t
    …
    xs = args t
    …
  in
    …
```

```
data TermF a
  = Var String
  | Abs String a
  | App a a
```

```
data ArgsF a
  = Var String
  | Abs String Term
  | App a Term
```

```
data NameF a
  = Var a
  | Abs a Term
  | App Term Term
```

fmap

fold

traverse

Are we done?

"Once you've decided to throw arbitrary unrolling of fixpoints into the language of types, the toothpaste has escaped from the tube and you have to resort to extreme cleverness to put it back in again."

-Reviewer A

- Type soundness

- Decidable typechecking

- Type soundness
    - Full F_ω^μ enjoys type soundness.
- Decidable typechecking

- Type soundness
  - Full F_{ω}^{μ} enjoys type soundness.
- Decidable typechecking
  - A fragment enjoys decidable typechecking.

Decidable fragment

$$\mu_\kappa : (\kappa \to \kappa) \to \kappa$$

Decidable fragment

μ_{*} : `(* → *) → *`

$\mu_{* \to *}$: `((* → *) → (* → *)) → (* → *)`

$\mu_{(* \to *) \to *}$: `(((* → *) → *) → ((* → *) → *)) →`
`(* → *) → *`

…

Decidable fragment

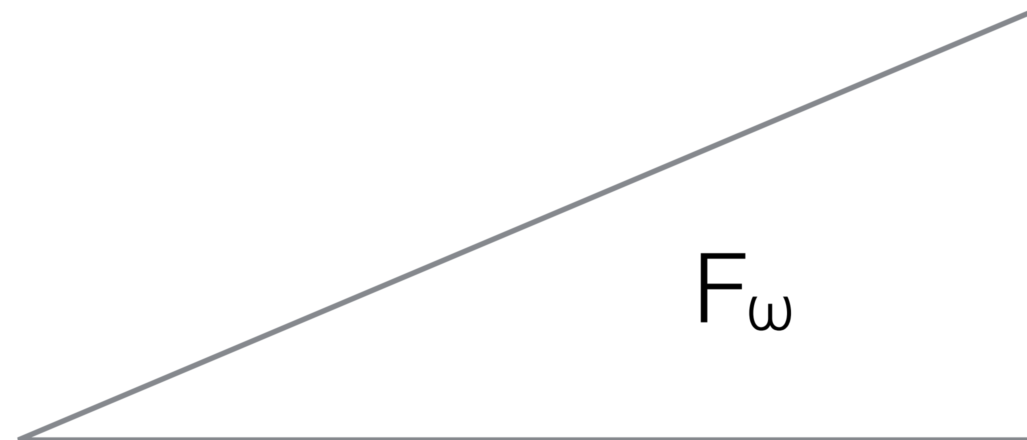
$\mu_* : (* \to *) \to *$

Decidable fragment

```
μ* : (* → *) → *
```

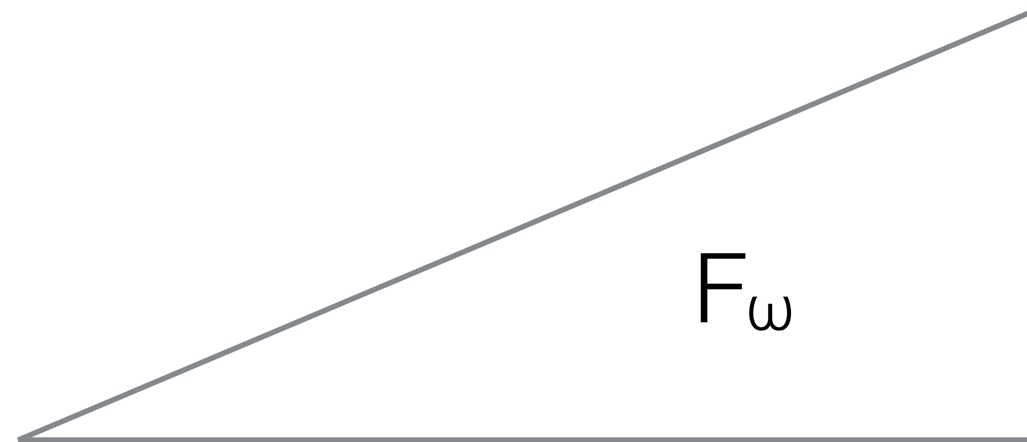
– Berarducci trees of types in this fragment
are regular (self-similar) trees

Typechecking



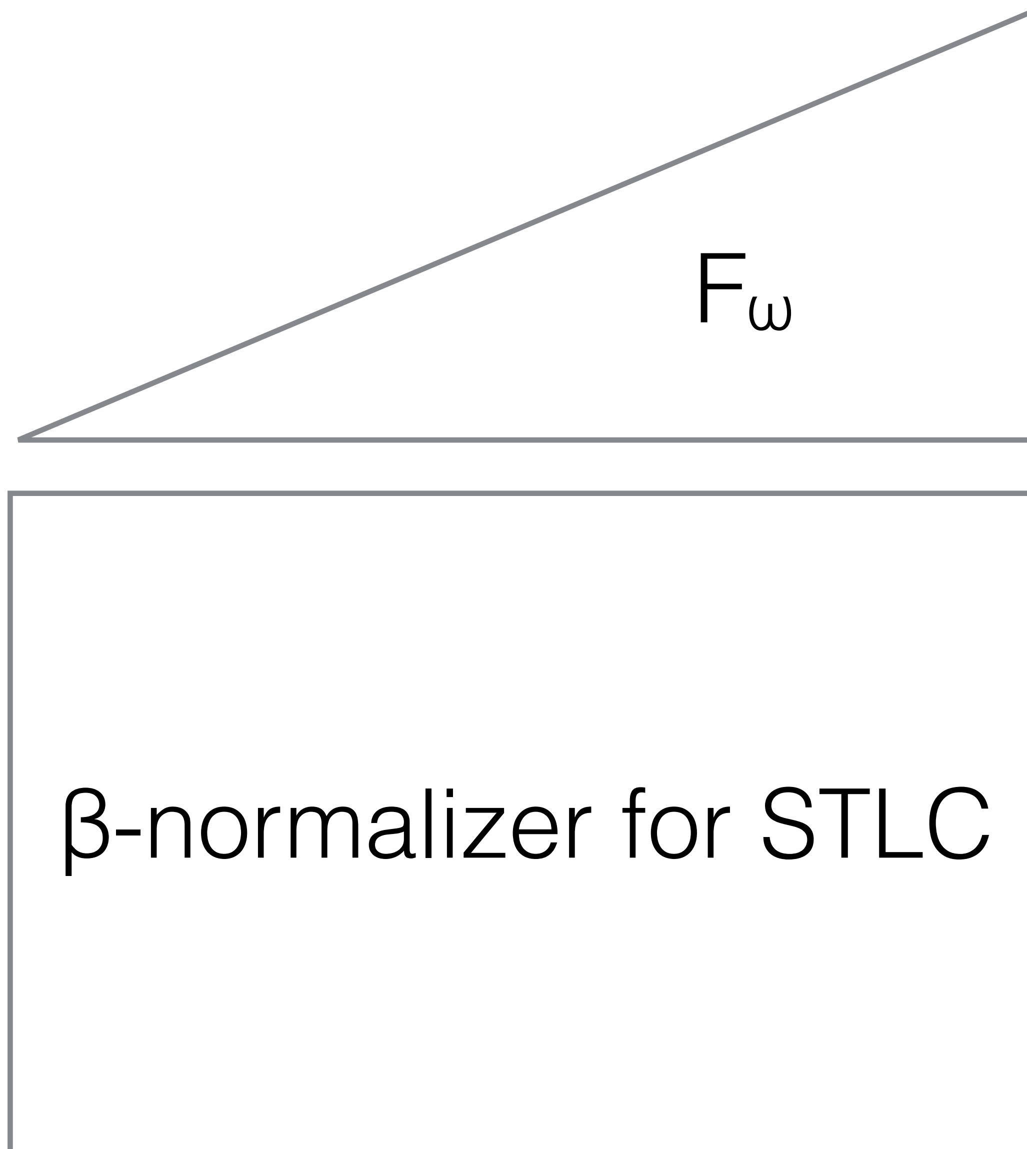
equality test for types

Typechecking

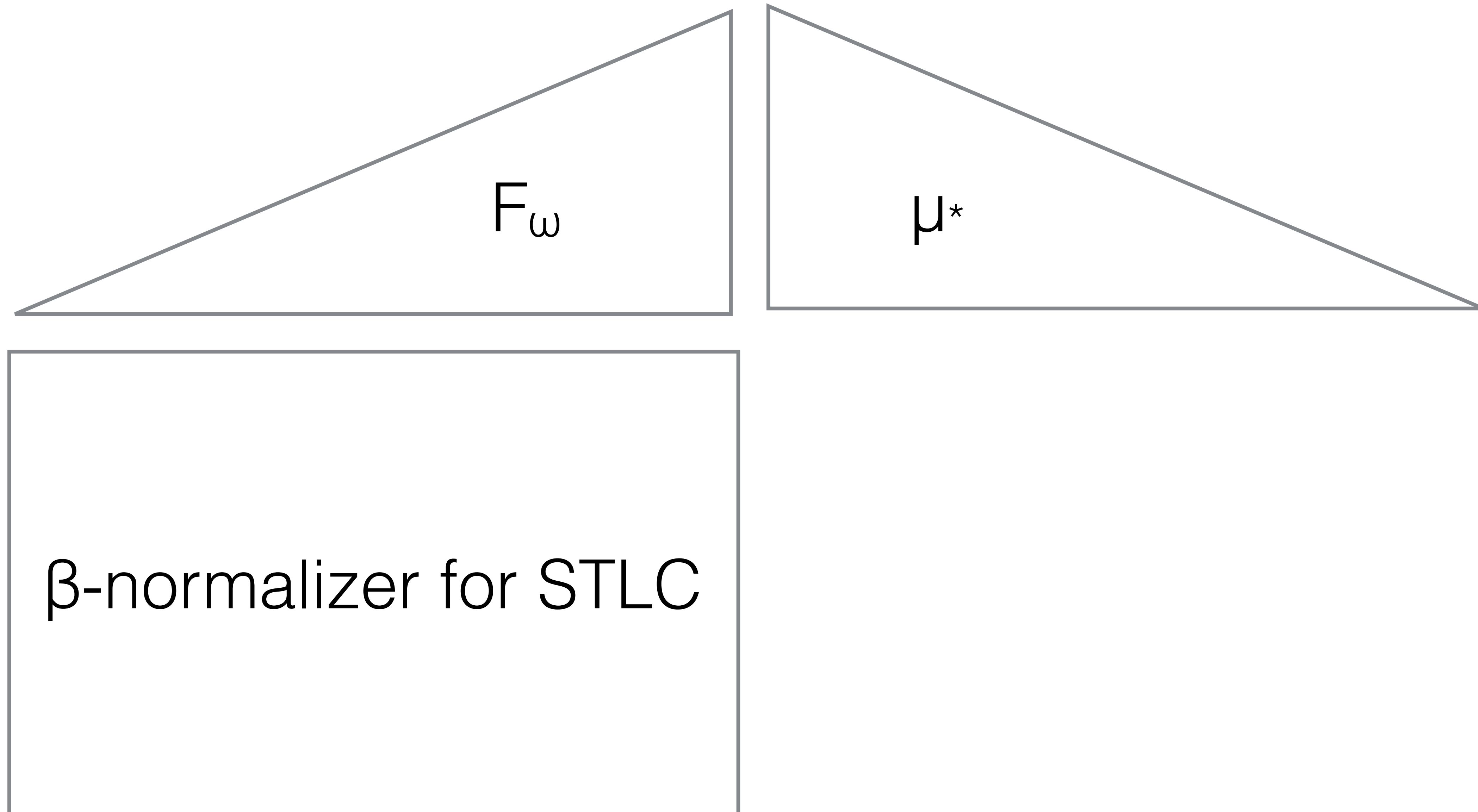


β-equivalence test
for STLC

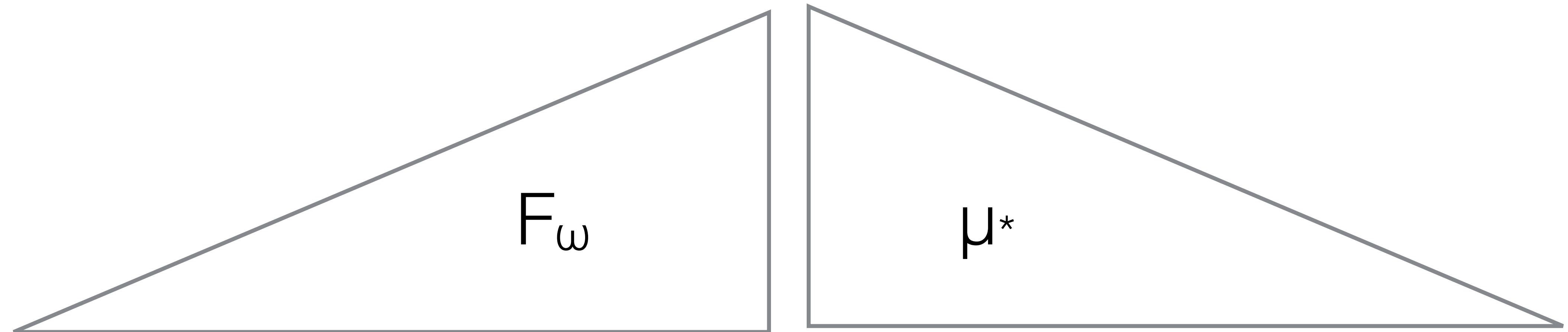
Typechecking



Typechecking

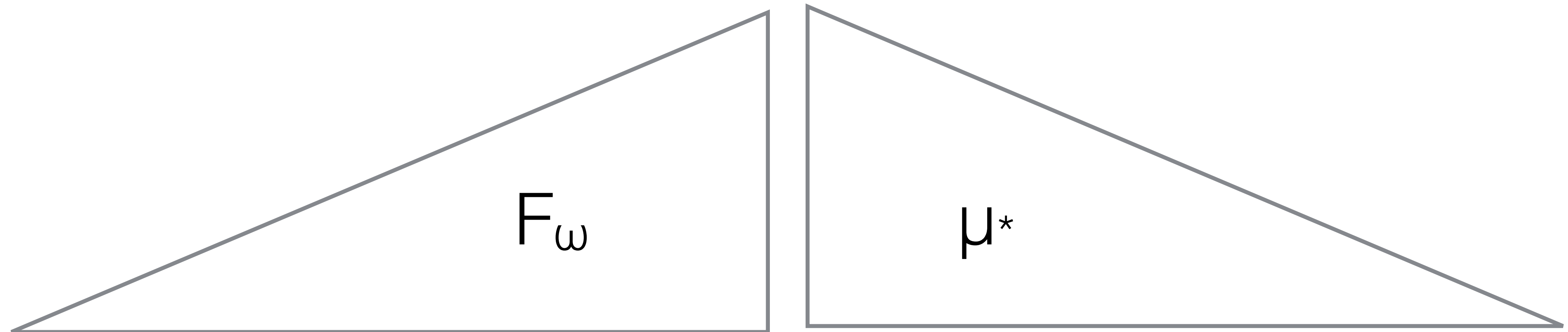


Typechecking



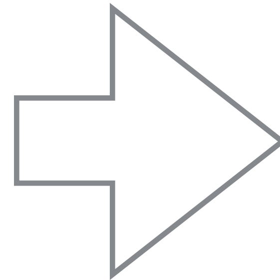
equality test for **infinite** types

Typechecking

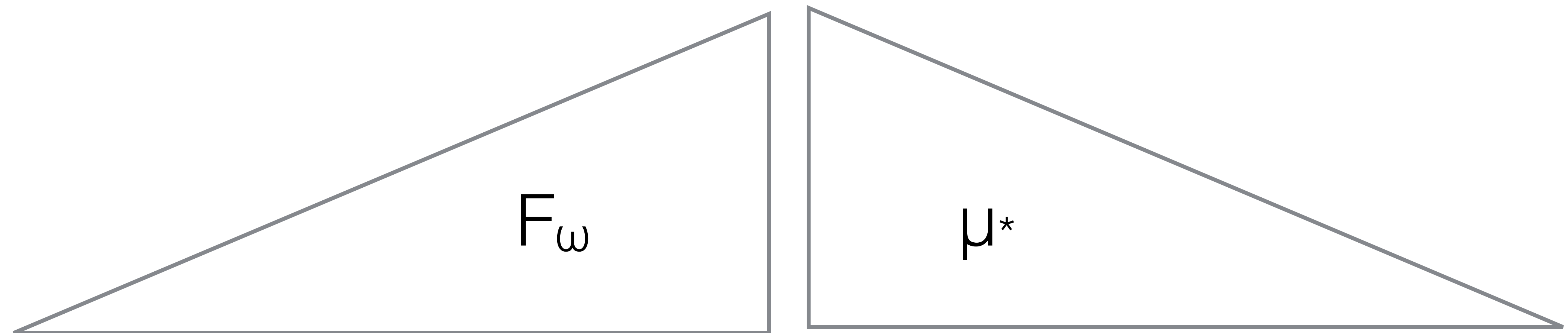


equality test for **infinite** types

β-normalize types
with μ_* uninterpreted

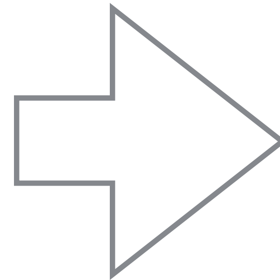


Typechecking



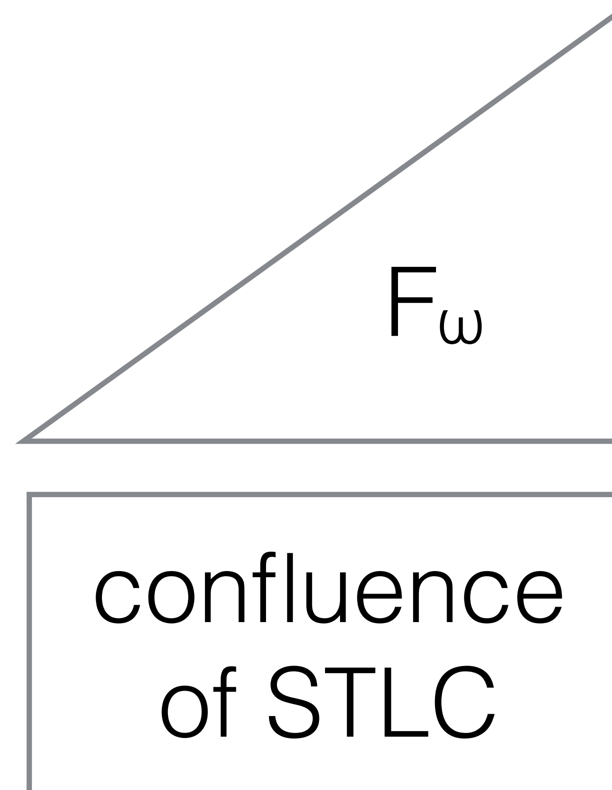
equality test for **infinite** types

β-normalize types with μ_* uninterpreted

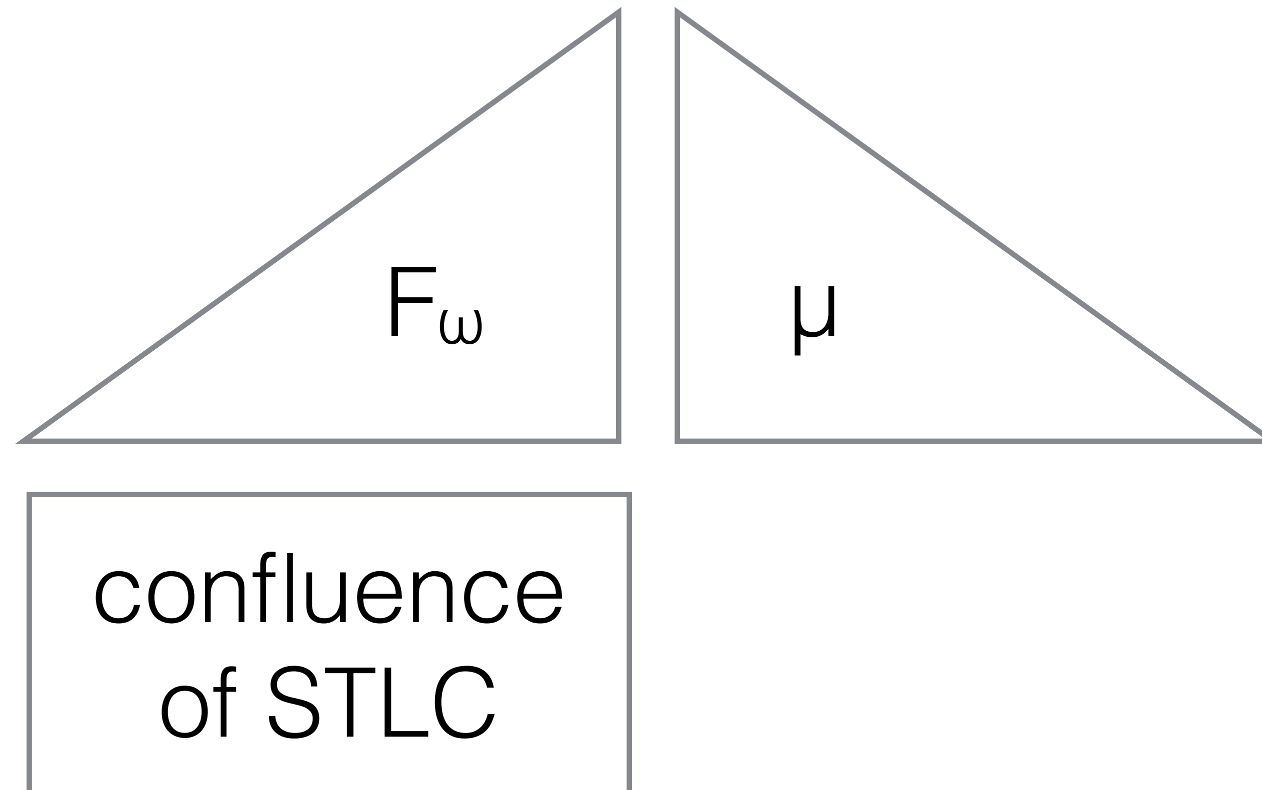


Henglein's algorithm

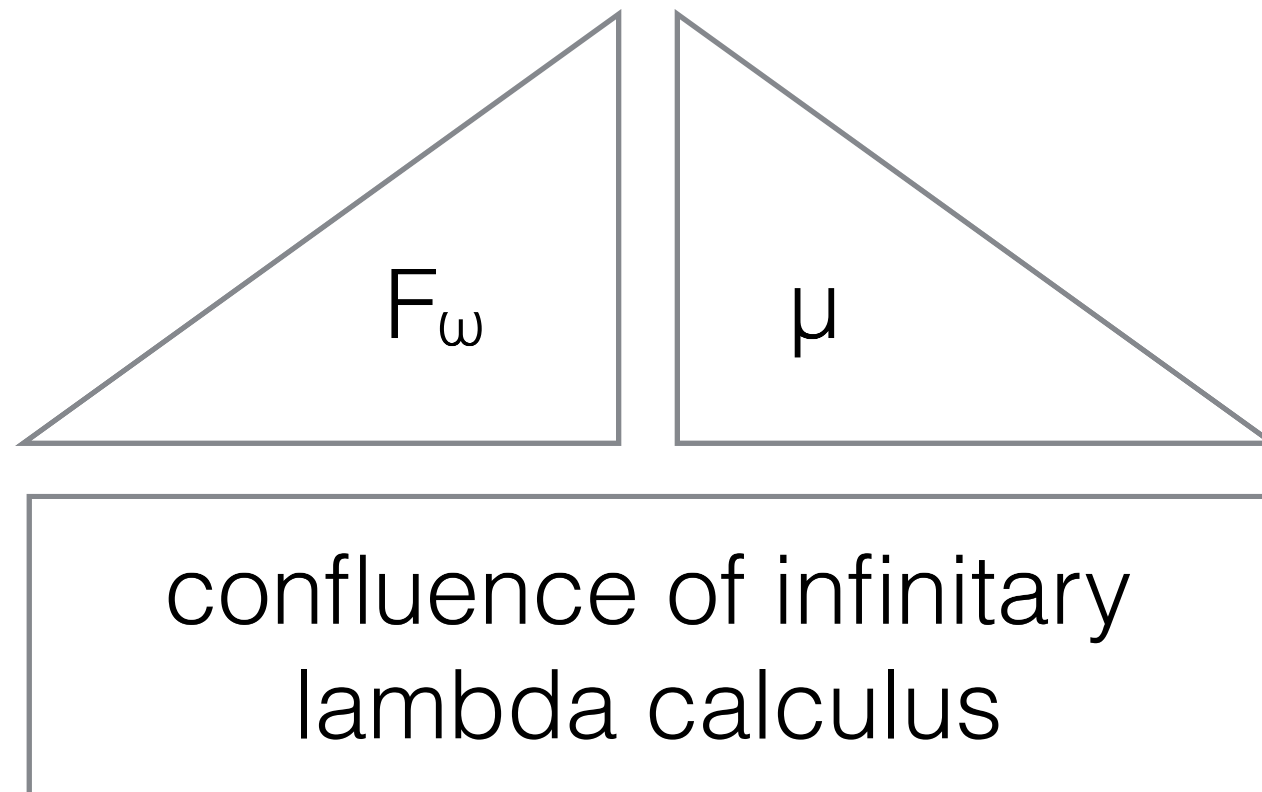
Type soundness



Type soundness



Type soundness



- Half the toothpaste is back in the tube
- What's next? Typechecking for full F^{μ}_{ω} ?

Closely related problems

- Equality of types constructed with higher-ranked
$\mu_\kappa : (\kappa \to \kappa) \to \kappa$

- Equivalence of higher-order *recursive schemes*
(i. e., mutually recursive types)

- Böhm-tree equivalence of λY-calculus (i. e., PCF without primitive operators)

Closely related problems

- Equality of types constructed with higher-ranked
  $\mu_\kappa : (\kappa \to \kappa) \to \kappa$

- Equivalence of higher-order *recursive schemes*
  (i. e., mutually recursive types)
  - Decidability is a long-standing well-known open problem in higher-order model checking

- Böhm-tree equivalence of λY-calculus (i. e., PCF without primitive operators)

Closely related problems

- Equality of types constructed with higher-ranked
  `μκ : (κ → κ) → κ`

- Equivalence of higher-order *recursive schemes*
  (i. e., mutually recursive types)

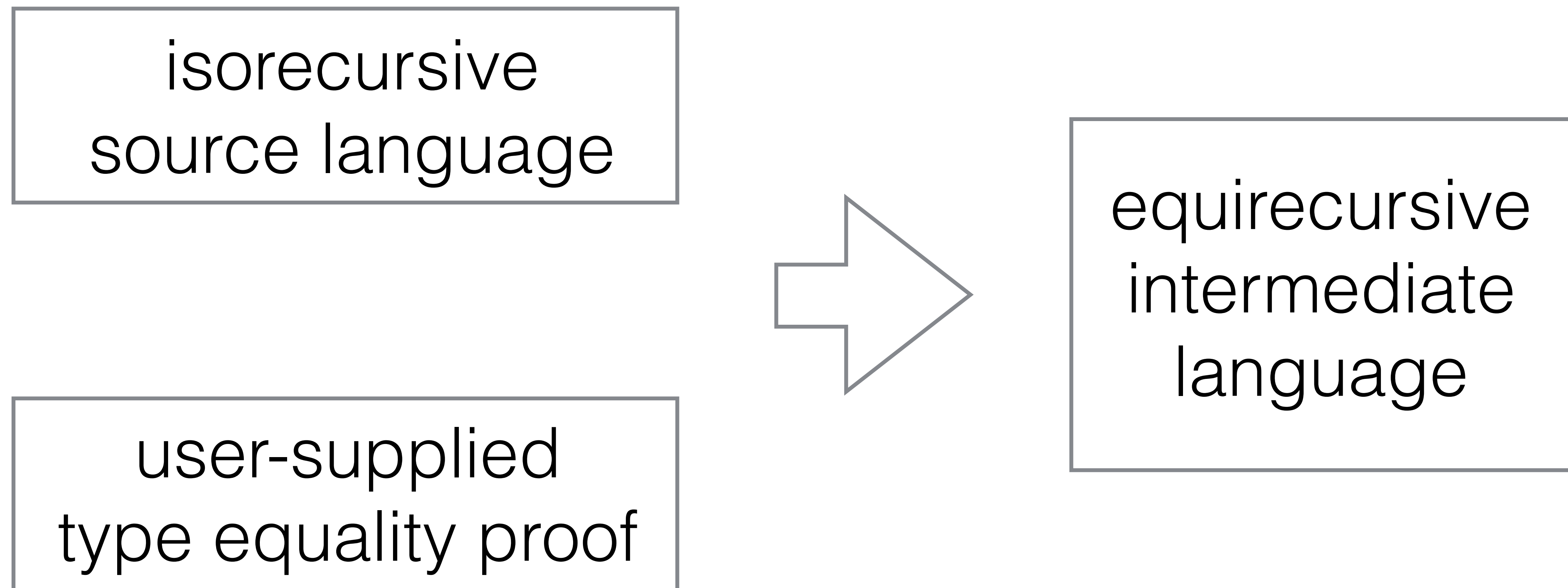
  - Decidability is a long-standing well-known open problem in higher-order model checking

- Böhm-tree equivalence of λY-calculus (i. e., PCF without primitive operators)

  - Is PCF without primitive operators Turing complete?

Work around decidability issue

- Reviewer A suggests asking the programmer for help



In the paper

- ☑ Tyranny of the dominant functor
- ☑ System F^{μ}_{ω} to save DGP from tyranny
- Proofs (technical report version)
- Logical relation to generate traversable functors in F^{μ}_{ω}