Polymorphic Functions with Set-Theoretic Types

Giuseppe Castagna

CNRS, Laboratoire Preuves, Programmes et Systèmes,
Université Paris Diderot, Paris, France.

Based on joint work with
Kim Nguyễn, Zhiwu Xu,
Pietro Abate, Hyeonsung Im, Sergueï Lenglet, Luca Padovani

(This work was presented at **POPL '14** and **POPL '15**)

Outline

Motivations and goals

i.e., why unions, intersections, and negations of types are useful (and not just for XML)

Set-theoretic types for classic data structures

Red-black trees are balanced binary search trees that must satisfy 4 invariants:

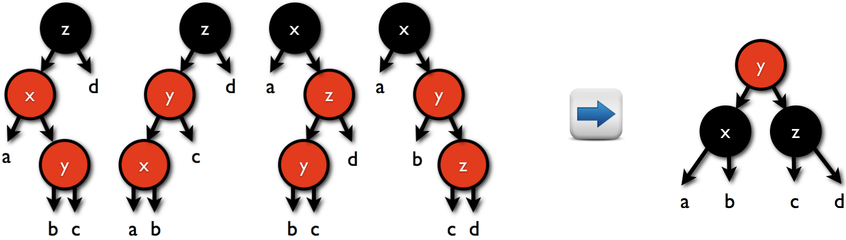
1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from root to a leaf contains the same number of black nodes

Set-theoretic types for classic data structures

Red-black trees are balanced binary search trees that must satisfy 4 invariants:

1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from root to a leaf contains the same number of black nodes

The key to implement `insert` is the function `balance` which transforms an *unbalanced tree*, into a *valid red tree* (as long as a, b, c, and d are valid):

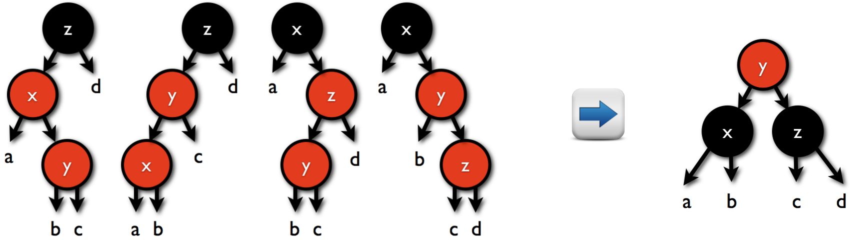


Set-theoretic types for classic data structures

Red-black trees are balanced binary search trees that must satisfy 4 invariants:

1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from root to a leaf contains the same number of black nodes

The key to implement `insert` is the function `balance` which transforms an *unbalanced tree*, into a *valid red tree* (as long as a, b, c, and d are valid):



In ML-like languages this yields a simple pattern-matching implementation:
[due to Okasaki: *Purely Functional Data Structures*, Cambridge Univ Press, 1998]

The code as written in Okasaki's book

```
type α RBtree =
   | Leaf
   | Red( α , RBtree , RBtree)
   | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

Notice that ML types *do not* enforce the invariants of the previous slide

```
type α RBtree =
    | Leaf
    | Red( α , RBtree , RBtree)
    | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

ML needs extra auxiliary functions *and* GADTs to enforce these invariants.

```
type α RBtree =
   | Leaf
   | Red( α , RBtree , RBtree)
   | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

In set-theoretic types these functions are straightforwardly typed **as they are**

```
type α RBtree =
   | Leaf
   | Red( α , RBtree , RBtree)
   | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
        if x < y  then balance c( y, (ins a), b ) else
        if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

In set-theoretic types these functions are straightforwardly typed **as they are**

① Write the correct type definitions

```
type α RBtree =
    | Leaf
    | Red( α , RBtree , RBtree)
    | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
      | Leaf -> Red(x,Leaf,Leaf)
      | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

In set-theoretic types these functions are straightforwardly typed **as they are**

① Write the correct type definitions

```
type α RBtree =
    | Leaf
    | Red( α , RBtree , RBtree)
    | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

In set-theoretic types these functions are straightforwardly typed **as they are**

```
type α RBtree =
   | Leaf
   | Red( α , RBtree , RBtree)
   | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

(1) Write the correct type definitions

(2) Add type annotations to function definitions

In set-theoretic types these functions are straightforwardly typed **as they are**

```
type α RBtree =
   | Leaf
   | Red( α , RBtree , RBtree)
   | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

(1) Write the correct type definitions

(2) Add type annotations to function definitions

①

```
type α RBtree =
   | Leaf
   | Red( α , RBtree , RBtree)
   | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

①

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree)|(RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree)|(RBtree,Wrong) )

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

②

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree)|(RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree)|(RBtree,Wrong) )

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
        if x < y  then balance c( y, (ins a), b ) else
        if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

②

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree)|(RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree)|(RBtree,Wrong) )

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
      | Leaf -> Red(x,Leaf,Leaf)
      | c(y,a,b) as z ->
          if x < y  then balance c( y, (ins a), b ) else
          if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

②

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree)|(RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree)|(RBtree,Wrong) )

let balance: (Unbal → Rtree) & ( (β\Unbal) → (β\Unbal) ) =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree) → Btree =
 function ( x , t ) ->
  let ins:(Leaf → Rtree) & (Btree → RBtree\Leaf) & (Rtree → Rtree|Wrong)=
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

①+②

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree) | (RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree) | (RBtree,Wrong) )

let balance: (Unbal → Rtree) & ( (β\Unbal) → (β\Unbal) ) =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree) → Btree =
 function ( x , t ) ->
  let ins:(Leaf → Rtree) & (Btree → RBtree\Leaf) & (Rtree → Rtree|Wrong)=
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

①+②

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree) | (RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree) | (RBtree,Wrong) )

let balance: (Unbal → Rtree) & ( (β\Unbal) → (β\Unbal) ) =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree) → Btree =
 function ( x , t ) ->
  let ins:(Leaf → Rtree) & (Btree → RBtree\Leaf) & (Rtree → Rtree|Wrong)=
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

- recursive types

①+②

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree) | (RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree) | (RBtree,Wrong) )

let balance: (Unbal → Rtree) & ( (β\Unbal) → (β\Unbal) ) =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree) → Btree =
 function ( x , t ) ->
  let ins: (Leaf → Rtree) & (Btree → RBtree\Leaf) & (Rtree → Rtree|Wrong)=
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
        if x < y  then balance c( y, (ins a), b ) else
        if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

- recursive types
- set-theoretic types

①+②

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree) | (RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree) | (RBtree,Wrong) )

let balance: (Unbal → Rtree) & ( (β\Unbal) → (β\Unbal) ) =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree) → Btree =
 function ( x , t ) ->
  let ins: (Leaf → Rtree) & (Btree → RBtree\Leaf) & (Rtree → Rtree | Wrong) =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
        if x < y  then balance c( y, (ins a), b ) else
        if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

- recursive types
- set-theoretic types
- polymorphic functions

①+②

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree) | (RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree) | (RBtree,Wrong) )

let balance: (Unbal → Rtree) & ( (β\Unbal) → (β\Unbal) ) =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree) → Btree =
 function ( x , t ) ->
  let ins:(Leaf → Rtree) & (Btree → RBtree\Leaf) & (Rtree → Rtree|Wrong)=
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

- recursive types
- set-theoretic types
- polymorphic functions

A simpler example of the same pattern

A motivating example in Haskell [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

A motivating example in Haskell [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |   _ -> x
```

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |   _ -> x
```

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |   _ -> x
```

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |   _ -> x
```

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          | _   -> x
```

type-case

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
              | [] -> []
              | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           | _   -> x
```

type-case

boolean type connectives

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an Int it returns a Bool; when applied to an argument that is not an Int it returns a result *of the same type*.

A motivating example in Haskell (almost) [no XML]

```
map :: (α→β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          | _ -> x
```

type variables

type-case

boolean type connectives

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |   _ -> x
```

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

> Common pattern for functional data structures: red-black trees balancing; ZDD operations; XML nodes modification

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |   _ -> x
```

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

The combination of type-case and intersections yields statically typed dynamic overloading.

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |   _ -> x
```

This example as a yardstick. I want to define a language that:

1. Can define both `map` and `even`

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
           | [] -> []
           | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |   _ -> x
```

This example as a yardstick. I want to define a language that:

1. Can define both `map` and `even`
2. Can *check* the types specified in the signature

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |   _ -> x
```

This example as a yardstick. I want to define a language that:

1. Can define both `map` and `even`
2. Can *check* the types specified in the signature
3. Can *deduce* the type of the partial application `map even`

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |   _ -> x
```

This example as a yardstick. I want to define a language that:

1. Can define both `map` and `even`
2. Can *check* the types specified in the signature
3. **Can deduce the type of the partial application `map even`**

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |   _ -> x
```

This example a… k. I want to define a language that:

Tough!

1. Can define … and even
2. Can *check* the … types specified in the signature
3. **Can deduce the type of the partial application map even**

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
             | [] -> []
             | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
            | Int -> (x `mod` 2) == 0
            |   _ -> x
```

We expect **map even** to have the following type:

$$
\begin{array}{l}
(\,[\,\mathtt{Int}\,] \to [\,\mathtt{Bool}\,]\,) \wedge \\
(\,[\,\alpha\backslash\mathtt{Int}\,] \to [\,\alpha\backslash\mathtt{Int}\,]\,) \wedge \\
(\,[\,\alpha\vee\mathtt{Int}\,] \to [\,(\alpha\backslash\mathtt{Int})\vee\mathtt{Bool}\,]\,)
\end{array}
$$

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x ‘mod‘ 2) == 0
          |   _ -> x
```

We expect **map even** to have the following type:

$([\mathtt{Int}] \to [\mathtt{Bool}]) \wedge$ int lists are transformed into bool lists

$([\alpha\backslash\mathtt{Int}] \to [\alpha\backslash\mathtt{Int}]) \wedge$ lists w/o ints return the same type

$([\alpha\vee\mathtt{Int}] \to [(\alpha\backslash\mathtt{Int})\vee\mathtt{Bool}])$ ints in the arg. are replaced by bools

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
              | [] -> []
              | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
             | Int -> (x `mod` 2) == 0
             |   _ -> x
```

We expect **map even** to have the following type:

$$
\begin{array}{ll}
(\,[\mathtt{Int}] \to [\mathtt{Bool}]\,) \wedge & \text{int lists are transformed into bool lists} \\
(\,[\alpha\backslash\mathtt{Int}] \to [\alpha\backslash\mathtt{Int}]\,) \wedge & \text{lists w/o ints return the same type} \\
(\,[\alpha\vee\mathtt{Int}] \to [(\alpha\backslash\mathtt{Int})\vee\mathtt{Bool}]\,) & \text{ints in the arg. are replaced by bools}
\end{array}
$$

Difficult because of expansion: needs *a set of type substitutions* — rather than just one— to unify the domain and the argument types.

Formal framework

i.e., all the gory details you do not want the programmer to ever know

Formal calculus

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\ ?\ e : e$
Types	t	$::=$	$B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Formal calculus

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}x.e \mid e{\in}t\,?\,e : e$
Types	t	$::=$	$B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Expressions include:

Formal calculus

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i} x.e \mid e{\in}t\,?\,e : e$
Types	t	$::=$	$B \mid t\to t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Expressions include:

A type-case:

- abstracts regular type patterns
- makes dynamic overloading possible

Formal calculus

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$
Types	t	$::=$	$B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Expressions include:

A type-case:

- abstracts regular type patterns
- makes dynamic overloading possible

Explicitly-typed functions:

- Needed by the type-case [e.g. $\mu f.\lambda x.f{\in}(\mathbb{1}{\to}\mathtt{Int})\,?\,\mathtt{true} : 42$]
- More expressive with the result type (parameter type not enough)

$\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$: well typed if for all $i{\in}I$ from $x : s_i$ we can deduce $e : t_i$.

Formal calculus

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

Types $t \quad ::= \quad B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Types may be **recursive** and have a **set-theoretic** interpretation:

Formal calculus

Exprs	e	::=	$x \mid ee \mid \lambda^{\wedge_{i\in I}s_i\to t_i}x.e \mid e{\in}t\,?\,e:e$
Types	t	::=	$B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \boldsymbol{\alpha}$

Types may be **recursive** and have a **set-theoretic** interpretation:

Constructors: $[\![\texttt{Int}]\!]{=}\{0,1,-1,...\}$. $[\![s \to t]\!]$ = all λ-abstractions that applied to arguments in $[\![s]\!]$ return only results in $[\![t]\!]$.

Formal calculus

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$
Types	t	$::=$	$B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Types may be **recursive** and have a **set-theoretic** interpretation:

Constructors: $[\![\texttt{Int}]\!] = \{0, 1, -1, ...\}$. $[\![s \to t]\!]$ = all λ-abstractions that applied to arguments in $[\![s]\!]$ return only results in $[\![t]\!]$.

Connectives have the corresponding set-theoretic interpretation:

$$[\![s{\vee}t]\!] = [\![s]\!] \cup [\![t]\!] \qquad [\![s{\wedge}t]\!] = [\![s]\!] \cap [\![t]\!] \qquad [\![\neg t]\!] = [\![\mathbb{1}]\!] \setminus [\![t]\!]$$

Formal calculus

Exprs $e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

Types $t ::= B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Types may be **recursive** and have a **set-theoretic** interpretation:

Constructors: $[\![\texttt{Int}]\!]{=}\{0,1,-1,\ldots\}$. $[\![s \to t]\!]$ = all λ-abstractions that applied to arguments in $[\![s]\!]$ return only results in $[\![t]\!]$.

Connectives have the corresponding set-theoretic interpretation:

$$[\![s{\vee}t]\!] = [\![s]\!] \cup [\![t]\!] \qquad [\![s{\wedge}t]\!] = [\![s]\!] \cap [\![t]\!] \qquad [\![\neg t]\!] = [\![\mathbb{1}]\!] \setminus [\![t]\!]$$

Subtyping:

- it is *defined* as set-containment: $s \leq t \stackrel{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$;

Formal calculus

$$\textbf{Exprs}\quad e \ ::= \ x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i} x.e \mid e\in t\,?\,e : e$$

$$\textbf{Types}\quad t \ ::= \ B \mid t\to t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$$

Types may be **recursive** and have a **set-theoretic** interpretation:

Constructors: $[\![\texttt{Int}]\!]=\{0,1,-1,\ldots\}$. $[\![s\to t]\!]$ = all λ-abstractions that applied to arguments in $[\![s]\!]$ return only results in $[\![t]\!]$.

Connectives have the corresponding set-theoretic interpretation:

$$[\![s\vee t]\!] = [\![s]\!]\cup[\![t]\!] \qquad [\![s\wedge t]\!] = [\![s]\!]\cap[\![t]\!] \qquad [\![\neg t]\!] = [\![\mathbb{1}]\!]\setminus[\![t]\!]$$

Subtyping with type variables:

- it is *defined* as set-containment: $s\leq t \overset{def}{\iff} [\![s]\!]\subseteq[\![t]\!]$;
- it is such that forall type-substitutions σ: $s\leq t \ \Rightarrow\ s\sigma\leq t\sigma$;
- it is decidable. [ICFP2011].

Formal calculus: **new stuff**

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$
Types	t	$::=$	$B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Polymorphic functions.

Formal calculus

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$
Types	t	$::=$	$B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Polymorphic functions: The novelty of this work is that type variables can occur in the *interfaces*.

Formal calculus: **new stuff**

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$
Types	t	$::=$	$B \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Polymorphic functions: The novelty of this work is that type variables can occur in the *interfaces*.

- $\lambda^{\alpha\to\alpha} x.x$ polymorphic identity
- $\lambda^{(\alpha\to\beta)\wedge\alpha\to\beta} x.xx$ auto-application

Formal calculus: **new stuff**

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

Types $t \quad ::= \quad B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Polymorphic functions: The novelty of this work is that type variables can occur in the *interfaces*.

- $\lambda^{\alpha\to\alpha}x.x$ — polymorphic identity
- $\lambda^{(\alpha\to\beta)\wedge\alpha\to\beta}x.xx$ — auto-application

Meaning: types obtained by subsumption *and by instantiation*

- $\lambda^{\alpha\to\alpha}x.x : \mathbb{0} \to \mathbb{1}$ — subsumption
- $\lambda^{\alpha\to\alpha}x.x : \neg\texttt{Int}$ — subsumption
- $\lambda^{\alpha\to\alpha}x.x : \texttt{Int} \to \texttt{Int}$ — instantiation
- $\lambda^{\alpha\to\alpha}x.x : \texttt{Bool} \to \texttt{Bool}$ — instantiation

Formal calculus: **new stuff**

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

Types $t \quad ::= \quad B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Problem

Define an explicitly typed, polymorphic calculus
with intersection types and dynamic type-case

Formal calculus: **new stuff**

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I}s_i\to t_i}x.e \mid e{\in}t\,?\,e : e$

Types $t \quad ::= \quad B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Problem

Define an explicitly typed, polymorphic calculus with intersection types and dynamic type-case

Formal calculus: **new stuff**

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$
Types	t	$::=$	$B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Problem

Define an explicitly typed, polymorphic calculus
with intersection types and dynamic type-case

Four simple points to show why dealing with this blend is quite problematic

1. Polymorphism needs instantiation:

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\mathtt{Int}/\alpha\}$:

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\mathtt{Int}/_{\alpha}\}$:

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

2. Type-case needs explicit relabeling:

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42 \quad \in \mathtt{Int}\to\mathtt{Int}$

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\mathsf{true} \notin \mathtt{Int}\to\mathtt{Int}$

Interfaces determine λ-abstractions's types [intrinsic semantics]

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\mathtt{Int}/_{\alpha}\}$:

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

2. Type-case needs explicit relabeling:

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42 \in \mathtt{Int}\to\mathtt{Int}$ $\quad\rightsquigarrow \lambda^{\mathtt{Int}\to\mathtt{Int}}y.42$

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\mathtt{true} \notin \mathtt{Int}\to\mathtt{Int}$ $\quad\rightsquigarrow \lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.\mathtt{true}$

Interfaces determine λ-abstractions's types [intrinsic semantics]

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\texttt{Int}/_{\alpha}\}$:

$$(\lambda^{\texttt{Int}\to\texttt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

2. Type-case needs explicit relabeling:

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42 \in \texttt{Int}\to\texttt{Int}$ $\quad\rightsquigarrow \lambda^{\texttt{Int}\to\texttt{Int}}y.42$

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\text{true} \notin \texttt{Int}\to\texttt{Int}$ $\quad\rightsquigarrow \lambda^{\texttt{Bool}\to\texttt{Bool}}y.\text{true}$

Interfaces determine λ-abstractions's types [intrinsic semantics]

3. Relabeling must be applied also on function bodies:

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\mathtt{Int}/_{\alpha}\}$:

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

2. Type-case needs explicit relabeling:

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42 \in \mathtt{Int}\to\mathtt{Int}$ $\quad\leadsto \lambda^{\mathtt{Int}\to\mathtt{Int}}y.42$

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\mathtt{true} \notin \mathtt{Int}\to\mathtt{Int}$ $\quad\leadsto \lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.\mathtt{true}$

Interfaces determine λ-abstractions's types [intrinsic semantics]

3. Relabeling must be applied also on function bodies:

A "daffy" definition of identity:

$$(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\mathtt{Int}/_{\alpha}\}$:

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

2. Type-case needs explicit relabeling:

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42 \in \mathtt{Int}\to\mathtt{Int}$ $\rightsquigarrow \lambda^{\mathtt{Int}\to\mathtt{Int}}y.42$

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\mathsf{true} \notin \mathtt{Int}\to\mathtt{Int}$ $\rightsquigarrow \lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.\mathsf{true}$

Interfaces determine λ-abstractions's types [intrinsic semantics]

3. Relabeling must be applied also on function bodies:

A "daffy" definition of identity:

$$(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

To apply it to 42, relabeling the outer λ by $\{\mathtt{Int}/_{\alpha}\}$ does not suffice:

$$(\lambda^{\alpha\to\alpha}y.42)42$$

is not well typed. The body must be relabeled as well, by applying the $\{\mathtt{Int}/_{\alpha}\}$ yielding: $(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.42)42$

4. Relabeling the body is not always so straightforward:

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$\texttt{Int}\to\texttt{Int} \qquad \texttt{Bool}\to\texttt{Bool}$$

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$$

So it has their intersection.

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$$

So it has their intersection.
We can feed the identity $\lambda^{\alpha\to\alpha}x.x$ to a function which expects an argument of the type above. But *how do we relabel it?*

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$$

So it has their intersection.
We can feed the identity $\lambda^{\alpha\to\alpha}x.x$ to a function which expects an argument of the type above. But *how do we relabel it?*
Intuitively: apply $\{\mathtt{Int}/_{\alpha}\}$ and $\{\mathtt{Bool}/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$$

So it has their intersection.
We can feed the identity $\lambda^{\alpha\to\alpha}x.x$ to a function which expects an argument of the type above. But *how do we relabel it?*
Intuitively: apply $\{\texttt{Int}/_{\alpha}\}$ and $\{\texttt{Bool}/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

$$(\lambda^{\alpha\to\alpha}x.x)[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}] \quad\rightsquigarrow\quad \lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$$

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$$

So it has their intersection.
We can feed the identity $\lambda^{\alpha\to\alpha}x.x$ to a function which expects an argument of the type above. But *how do we relabel it?*
Intuitively: apply $\{\texttt{Int}/_{\alpha}\}$ and $\{\texttt{Bool}/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

$$(\lambda^{\alpha\to\alpha}x.x)[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}] \quad \rightsquigarrow \quad \lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$$

We applied a set of type substitutions: $t[\sigma_i]_{i\in I} = \bigwedge_{i\in I} t\sigma_i$

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$\mathtt{Int}\to\mathtt{Int} \qquad \mathtt{Bool}\to\mathtt{Bool}$$

So it has their intersection.
We can feed the the identity $\lambda^{\alpha\to\alpha}x.x$ to a function which expects an argument of the type above. But *how do we relabel it?*
Intuitively: apply $\{\mathtt{Int}/_{\alpha}\}$ and $\{\mathtt{Bool}/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

$$(\lambda^{\alpha\to\alpha}x.x)[\{\mathtt{Int}/_{\alpha}\},\{\mathtt{Bool}/_{\alpha}\}] \quad\rightsquigarrow\quad \lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.x$$

We applied a *set* of type substitutions: $t[\sigma_i]_{i\in I} = \bigwedge_{i\in I} t\sigma_i$.

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

Consider again the daffy identity $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$.
It also has type

$$(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$$

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

Consider again the daffy identity $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$.
It also has type

$$(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$$

Applying the set of substitutions $[\{\texttt{Int}/_{\alpha}\}, \{\texttt{Bool}/_{\alpha}\}]$ both to the interface and the body yields an ill-typed term:

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x)$$

4. Relabeling the body is not always so straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

Consider again the daffy identity $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$.
It also has type

$$(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$$

Applying the set of substitutions $[\{{}^{\mathtt{Int}}/_{\alpha}\}, \{{}^{\mathtt{Bool}}/_{\alpha}\}]$ both to the interface and the body yields an ill-typed term:

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x)$$

Let us see why it is not well typed

In order to type

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

In order to type

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \texttt{Int} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Int}$
2. $x : \texttt{Bool} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Bool}$

In order to type

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \texttt{Int} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Int}$
2. $x : \texttt{Bool} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Bool}$

Both fail because $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is not well typed

In order to type

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \texttt{Int} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Int}$
2. $x : \texttt{Bool} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Bool}$

Both fail because $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is not well typed

In order to type

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \texttt{Int} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Int}$
2. $x : \texttt{Bool} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Bool}$

Both fail because $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is not well typed

Key idea

The relabeling of the body *must change*
according to the type of the parameter

In order to type

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \texttt{Int} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Int}$
2. $x : \texttt{Bool} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Bool}$

Both fail because $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is not well typed

Key idea

The relabeling of the body *must change*
according to the type of the parameter

In our example with $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$ and $[\{\texttt{Int}/_{\alpha}\}, \{\texttt{Bool}/_{\alpha}\}]$:

In order to type

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \texttt{Int} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Int}$
2. $x : \texttt{Bool} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Bool}$

FAILED

Both fail because $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is not well typed

Key idea

The relabeling of the body *must change*
according to the type of the parameter

In our example with $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$ and $[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}]$:

In order to type

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \texttt{Int} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Int}$
2. $x : \texttt{Bool} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Bool}$

Both fail because $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is not well typed

Key idea

The relabeling of the body *must change*
according to the type of the parameter

In our example with $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$ and $[\{\texttt{Int}/_{\alpha}\}, \{\texttt{Bool}/_{\alpha}\}]$:

- $(\lambda^{\alpha\to\alpha}y.x)$ must be relabeled as $(\lambda^{\texttt{Int}\to\texttt{Int}}y.x)$ when $x : \texttt{Int}$;
- $(\lambda^{\alpha\to\alpha}y.x)$ must be relabeled as $(\lambda^{\texttt{Bool}\to\texttt{Bool}}y.x)$ when $x : \texttt{Bool}$

A new technique

Observation

This "dependent relabeling" is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

A new technique

Observation

This "dependent relabeling" is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

Our new technique: "lazy" relabeling of bodies.

- *Decorate λ-abstractions by sets of type-substitutions:*

A new technique

Observation

This "dependent relabeling" is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

Our new technique: "lazy" relabeling of bodies.

- *Decorate λ-abstractions by sets of type-substitutions:*
  To pass the daffy identity to a function that expects arguments of type $(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})$ first "lazily" relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/_{\alpha}\},\{\mathtt{Bool}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

A new technique

Observation

This "dependent relabeling" is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

Our new technique: "lazy" relabeling of bodies.

- *Decorate λ-abstractions by sets of type-substitutions:*
  To pass the daffy identity to a function that expects arguments of type $(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$ first "lazily" relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/_\alpha\},\{\mathtt{Bool}/_\alpha\}]} x.(\lambda^{\alpha\to\alpha} y.x)x)$$

- The decoration indicates that the function must be relabeled

A new technique

Observation

This "dependent relabeling" is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

Our new technique: "lazy" relabeling of bodies.

- *Decorate λ-abstractions by sets of type-substitutions:*
  To pass the daffy identity to a function that expects arguments of type $(\texttt{Int}{\to}\texttt{Int}) \wedge (\texttt{Bool}{\to}\texttt{Bool})$
  first "lazily" relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}]} x.(\lambda^{\alpha\to\alpha} y.x)x)$$

- The decoration indicates that the function must be relabeled
- The relabeling will be actually propagated to the body of the function at the moment of the reduction (lazy relabeling)

A new technique

Observation

This "dependent relabeling" is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

Our new technique: "lazy" relabeling of bodies.

- *Decorate λ-abstractions by sets of type-substitutions:*
  To pass the daffy identity to a function that expects arguments of type $(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$ first "lazily" relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

- The decoration indicates that the function must be relabeled
- The relabeling will be actually propagated to the body of the function at the moment of the reduction (lazy relabeling)
- The new decoration is statically used by the type system to ensure soundness.

Details follow, but remember we want to program in this language

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

No decorations: We do not want to oblige the programmer to write any explicit type substitution.

Details follow, but remember we want to program in this language

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

No decorations: We do not want to oblige the programmer to write any explicit type substitution.

The technical development will proceed as follows:

1. Define a calculus with explicit type-substitutions and decorated λ-abstractions.
2. Define an inference system that deduces where to insert explicit type-substitutions in a term of the language above
3. Define a compilation and execution technique thanks to which type substitutions are computed only when strictly necessary (in general, as efficient as a monomorphic execution).

Details follow, but remember we want to program in this language

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t \,?\, e : e$$

No decorations: We do not want to oblige the programmer to write any explicit type substitution.

The technical development will proceed as follows:

1. Define a calculus with explicit type-substitutions and decorated λ-abstractions.
2. Define an inference system that deduces where to insert explicit type-substitutions in a term of the language above
3. Define a compilation and execution technique thanks to which type substitutions are computed only when strictly necessary (in general, as efficient as a monomorphic execution).

Before proceeding we can already check our first yardstick:

$$\begin{array}{rcl} \texttt{even} & = & \lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\setminus\texttt{Int}\to\alpha\setminus\texttt{Int})} x\,.\, x{\in}\texttt{Int}\,?\,(x \bmod 2) = 0 : x \\ \texttt{map} & = & \mu m^{(\alpha\to\beta)\to[\alpha]\to[\beta]}\ f\,. \\ & & \qquad \lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell{\in}\texttt{nil}\,?\,\texttt{nil} : (f(\pi_1\ell), mf(\pi_2\ell)) \end{array}$$

A calculus with explicit type-substitutions

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Some examples:

- ✗ $(\lambda^{\alpha\to\alpha}x.x)42$
- ✓ $(\lambda^{\alpha\to\alpha}x.x)[\{\texttt{Int}/_{\alpha}\}]42$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\ ?\ e : e \mid e[\sigma_i]_{i\in I}$$

Some examples:

- ✗ $(\lambda^{\alpha\to\alpha} x.x)42$
- ✓ $(\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/_{\alpha}\}]42$
- ✓ $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/_{\alpha}\}]} x.x)42$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\ ?\ e : e \mid e[\sigma_i]_{i\in I}$$

Some examples:

- ✗ $(\lambda^{\alpha\to\alpha} x.x)42$
- ✓ $(\lambda^{\alpha\to\alpha} x.x)[\{\texttt{Int}/_{\alpha}\}]42$
- ✓ $(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/_{\alpha}\}]} x.x)42$
- ✗ $(\lambda^{\alpha\to\alpha} x.x)[\{\texttt{Bool}/_{\alpha}\}]42$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Some examples:

- ✗ $(\lambda^{\alpha\to\alpha} x.x)42$
- ✓ $(\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/_{\alpha}\}]42$
- ✓ $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/_{\alpha}\}]} x.x)42$
- ✗ $(\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Bool}/_{\alpha}\}]42$
- ✗ $(\lambda^{(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Int}} y.y3)(\lambda^{\alpha\to\alpha} x.x)$
- ✓ $(\lambda^{(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Int}} y.y3)((\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/_{\alpha}\}])$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Some examples:

- ✗ $(\lambda^{\alpha\to\alpha}x.x)42$
- ✓ $(\lambda^{\alpha\to\alpha}x.x)[\{\mathtt{Int}/_{\alpha}\}]42$
- ✓ $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/_{\alpha}\}]}x.x)42$
- ✗ $(\lambda^{\alpha\to\alpha}x.x)[\{\mathtt{Bool}/_{\alpha}\}]42$
- ✗ $(\lambda^{(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Int}}y.y3)(\lambda^{\alpha\to\alpha}x.x)$
- ✓ $(\lambda^{(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Int}}y.y3)((\lambda^{\alpha\to\alpha}x.x)[\{\mathtt{Int}/_{\alpha}\}])$
- ✓ $(\lambda^{((\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool}))\to t}y.e)((\lambda^{\alpha\to\alpha}x.x)[\{\mathtt{Int}/_{\alpha}\},\{\mathtt{Bool}/_{\alpha}\}])$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** *pushes the type substitutions into the decorations of the* λ*'s inside* e

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e\in t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & x \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}} x.e \\
(e_1 e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) \\
(e\in t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@[\sigma_j]_{j\in J}\in t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})
\end{array}$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$
\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & x \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}} x.e \\
(e_1 e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) \\
(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})
\end{array}
$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & x \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}} x.e \\
(e_1 e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) \\
(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})
\end{array}$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$: [Pushes σ's down into λ's]

$$x@[\sigma_j]_{j\in J} \stackrel{\text{def}}{=} x$$

$$(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} \stackrel{\text{def}}{=} \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}} x.e$$

$$(e_1 e_2)@[\sigma_j]_{j\in J} \stackrel{\text{def}}{=} (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J})$$

$$(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} \stackrel{\text{def}}{=} e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J}$$

$$(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} \stackrel{\text{def}}{=} e@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$
\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & x \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}} x.e \\
(e_1 e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) \\
(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})
\end{array}
$$

Notions of reduction:

$$
\begin{array}{rcl}
e[\sigma_j]_{j\in J} & \rightsquigarrow & e@[\sigma_j]_{j\in J} \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e)v & \rightsquigarrow & (e@[\sigma_j]_{j\in P})\{v/_x\} \qquad P = \{j{\in}J \mid \exists i{\in}I, \vdash v : t_i\sigma_j\} \\
v{\in}t\,?\,e_1 : e_2 & \rightsquigarrow & \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases}
\end{array}
$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e\in t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & x \\
(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}} x.e \\
(e_1 e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) \\
(e\in t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@[\sigma_j]_{j\in J}\in t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J})
\end{array}$$

Notions of reduction:

$$\begin{array}{rcll}
e[\sigma_j]_{j\in J} & \leadsto & e@[\sigma_j]_{j\in J} & \\
(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e)v & \leadsto & (e@[\sigma_j]_{j\in P})\{v/x\} & P = \{j\in J \mid \exists i\in I, \vdash v : t_i\sigma_j\} \\
v\in t\,?\,e_1 : e_2 & \leadsto & \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases} &
\end{array}$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$x@[\sigma_j]_{j\in J} \overset{\text{def}}{=} x$

$(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j \ldots \overset{\text{def}}{=} \lambda^{\wedge_{i\in I} t_i\to s_i} \ldots x.e$

$(e_1 e_2)@[\ldots$

$(e{\in}t\,?\,e_1 : e_2)@[\ldots : e_2@[\sigma_j]_{j\in J}$

$(e[\sigma_k]_{k\in K})@[\ldots$

Only keep the substitutions that make the type of the argument v match at least one input type of the interface

Notions of reduction:

$e[\sigma_j]_{j\in J} \rightsquigarrow e@[\sigma_j]_{j\in J}$

$(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e)v \rightsquigarrow (e@[\sigma_j]_{j\in P})\{{}^{v}\!/\!{}_{x}\}$ $\qquad P = \{j{\in}J \mid \exists i{\in}I, \vdash v : t_i\sigma_j\}$

$v{\in}t\,?\,e_1 : e_2 \rightsquigarrow \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases}$

Example

$$(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

Example

$$\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)z$$

Example

$$\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}]z$$

Example

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\mathtt{Int}}\!/\!_{\alpha}\},\{{}^{\mathtt{Bool}}\!/\!_{\alpha}\}]z)42$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} z.(\lambda^{\alpha\to\alpha} x.(\lambda^{\alpha\to\alpha} y.x)x)[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]z)42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha} x.(\lambda^{\alpha\to\alpha} y.x)x)[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]42$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]z)42$$

$$\rightsquigarrow\quad(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]42$$

$$\rightsquigarrow\quad(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)42$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]z)42$$

$$\rightsquigarrow\quad (\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]42$$

$$\rightsquigarrow\quad (\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)42$$

$$\rightsquigarrow\quad (\lambda^{\texttt{Int}\to\texttt{Int}}y.42)42$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} z.(\lambda^{\alpha\to\alpha} x.(\lambda^{\alpha\to\alpha} y.x)x)[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]z)42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha} x.(\lambda^{\alpha\to\alpha} y.x)x)[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]} x.(\lambda^{\alpha\to\alpha} y.x)x)42$$

$$\rightsquigarrow \quad (\lambda^{\texttt{Int}\to\texttt{Int}} y.42)42$$

no Bool here

Example

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})} z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]z)42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)42$$

$$\rightsquigarrow \quad (\lambda^{\mathtt{Int}\to\mathtt{Int}}y.42)42$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} z.(\lambda^{\alpha\to\alpha} x.(\lambda^{\alpha\to\alpha} y.x)x)[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}]z)42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha} x.(\lambda^{\alpha\to\alpha} y.x)x)[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}]42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}]} x.(\lambda^{\alpha\to\alpha} y.x)x)42$$

REJECTED

$$\rightsquigarrow \quad (\lambda^{\texttt{Int}\to\texttt{Int}} y.42)42 \qquad \equiv (((\lambda^{\alpha\to\alpha} y.x)x)@[\{\texttt{Int}/_{\alpha}\}])\{42/_{x}\}$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} z.(\lambda^{\alpha\to\alpha} x.(\lambda^{\alpha\to\alpha} y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]z)42$$

$$\rightsquigarrow\quad (\lambda^{\alpha\to\alpha} x.(\lambda^{\alpha\to\alpha} y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]42$$

$$\rightsquigarrow\quad (\lambda^{\alpha\to\alpha}_{[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]} x.(\lambda^{\alpha\to\alpha} y.x)x)42$$

$$\rightsquigarrow\quad (\lambda^{\texttt{Int}\to\texttt{Int}} y.42)42 \qquad \equiv (((\lambda^{\alpha\to\alpha} y.x)x)@[\{{}^{\texttt{Int}}/_{\alpha}\}])\{{}^{42}/_{x}\}$$

$$\rightsquigarrow\quad 42$$

Type system

$$(\mathit{subsumption})\ \frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2}$$

$$(\mathit{appl})\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$(\mathit{inst})\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\quad \sigma_j \sharp\, \Gamma$$

$$(\mathit{abstr})\ \frac{\Gamma, x : t_i\sigma_j \vdash e[\sigma_j] : s_i\sigma_j}{\Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\quad \begin{array}{l} i \in I \\ j \in J \end{array}$$

[plus the rules for type-case and variables]

Type system

$$(subsumption)\ \frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2}$$

$$(appl)\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$(inst)\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j} \quad \sigma_j \sharp\, \Gamma$$

$$(abstr)\ \frac{\Gamma, x : t_i\sigma_j \vdash e[\sigma_j] : s_i\sigma_j}{\Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j} \quad \begin{array}{l} i \in I \\ j \in J \end{array}$$

[plus the rules for type-case and variables]

Type system

$$(subsumption)\ \frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2}$$

$$(appl)\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$(inst)\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t\sigma_j}\ \sigma_j \sharp\, \Gamma$$

$$(abstr)\ \frac{\Gamma, x : t_i \quad \vdash e \quad : s_i}{\Gamma \vdash \lambda^{\wedge_{i \in I} t_i \to s_i} x.e : \bigwedge_{i \in I} \ t_i \ \to s_i}\ i \in I$$

[plus the rules for type-case and variables]

Type system

$$(\textit{subsumption})\ \frac{\Gamma \vdash e : t_1 \qquad t_1 \leq t_2}{\Gamma \vdash e : t_2}$$

$$(\textit{appl})\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$(\textit{inst})\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\ \sigma_j \sharp \Gamma$$

$$(\textit{abstr})\ \frac{\Gamma, x : t_i\sigma_j \vdash e[\sigma_j] : s_i\sigma_j}{\Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\ \begin{array}{l} i \in I \\ j \in J \end{array}$$

[plus the rules for type-case and variables]

Properties

Theorem (Subject Reduction)

For every term e and type t, if $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e be a well-typed closed term. If e is not a value, then there exists a term e' such that $e \rightsquigarrow e'$.

Properties

Theorem (Subject Reduction)

For every term e and type t, if $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e be a well-typed closed term. If e is not a value, then there exists a term e' such that $e \rightsquigarrow e'$.

Theorem

Let $\vdash_{BCD}$ be Barendregt, Coppo, and Dezani, typing, and $\lceil e \rceil$ the type erasure of e. If $\vdash_{BCD} a : t$, then $\exists e$ s.t. $\vdash e : t$ and $\lceil e \rceil = a$.

Properties

Theorem (Subject Reduction)

For every term e and type t, if $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e be a well-typed closed term. If e is not a value, then there exists a term e' such that $e \rightsquigarrow e'$.

Theorem

Let $\vdash_{BCD}$ be Barendregt, Coppo, and Dezani, typing, and $\lceil e \rceil$ the type erasure of e. If $\vdash_{BCD} a : t$, then $\exists e$ s.t. $\vdash e : t$ and $\lceil e \rceil = a$.

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Properties

Theorem (Subject Reduction)

For every term e *and type* t*, if* $\Gamma \vdash e : t$ *and* $e \rightsquigarrow e'$*, then* $\Gamma \vdash e' : t$*.*

Theorem (Progress)

Let e *be a well-typed closed term. If* e *is not a value, then there exists a term* e' *such that* $e \rightsquigarrow e'$*.*

Theorem

Let $\vdash_{BCD}$ *be Barendregt, Coppo, and Dezani, typing, and* $\lceil e \rceil$ *the type erasure of* e*. If* $\vdash_{BCD} a : t$*, then* $\exists e$ *s.t.* $\vdash e : t$ *and* $\lceil e \rceil = a$*.*

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \quad \mid e[\sigma_i]_{i\in I}$$

Properties

Theorem (Subject Reduction)

For every term e and type t, if $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e be a well-typed closed term. If e is not a value, then there exists a term e' such that $e \rightsquigarrow e'$.

Theorem

Let $\vdash_{BCD}$ be Barendregt, Coppo, and Dezani, typing, and $\lceil e \rceil$ the type erasure of e. If $\vdash_{BCD} a : t$, then $\exists e$ s.t. $\vdash e : t$ and $\lceil e \rceil = a$.

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t \,?\, e : e \mid \cancel{e[\sigma_i]_{i\in I}}$$

Properties

Theorem (Subject Reduction)

For every term e and type t, if $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e be a well-typed closed term. If e is not a value, then there exists a term e' such that $e \rightsquigarrow e'$.

Theorem

Let $\vdash_{BCD}$ be Barendregt, Coppo, and Dezani, typing, and $\lceil e \rceil$ the type erasure of e. If $\vdash_{BCD} a : t$, then $\exists e$ s.t. $\vdash e : t$ and $\lceil e \rceil = a$.

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid \not{e[\sigma_j]_{j\in J}}$$

satisfies the above theorem and is closed by reduction.

Properties

Theorem (Subject Reduction)

For every term e *and type* t*, if* $\Gamma \vdash e : t$ *and* $e \rightsquigarrow e'$*, then* $\Gamma \vdash e' : t$*.*

Theorem (Progress)

Let e *be a well-typed closed term. If* e *is not a value, then there exists a term* e' *such that* $e \rightsquigarrow e'$*.*

Theorem

Let $\vdash_{BCD}$ *be Barendregt, Coppo, and Dezani, typing, and* $\lceil e \rceil$ *the type erasure of* e*. If* $\vdash_{BCD} a : t$*, then* $\exists e$ *s.t.* $\vdash e : t$ *and* $\lceil e \rceil = a$*.*

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

satisfies the above theorem and is closed by reduction, too.

Properties

Theorem (Subject Reduction)

For every term e and type t, if $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e be a well-typed closed term. If e is not a value, then there exists a term e' such that $e \rightsquigarrow e'$.

Theorem

Let $\vdash_{BCD}$ be Barendregt, Coppo, and Dezani, typing, and $\lceil e \rceil$ the type erasure of e. If $\vdash_{BCD} a : t$, then $\exists e$ s.t. $\vdash e : t$ and $\lceil e \rceil = a$.

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

The first n terms ($n = 3, 4, 5$) form an explicitly-typed λ-calculus with intersection types subsuming BCD.

Properties

The definitions we gave:

$$\begin{array}{rcl} \texttt{even} & = & \lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\setminus\texttt{Int}\to\alpha\setminus\texttt{Int})}x\,.\,x{\in}\texttt{Int}\,?\,(x \bmod 2) = 0 : x \\ \texttt{map} & = & \mu m^{(\alpha\to\beta)\to[\alpha]\to[\beta]}\ f. \\ & & \qquad \lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell{\in}\texttt{nil}\,?\,\texttt{nil} : (f(\pi_1\ell), mf(\pi_2\ell)) \end{array}$$

are well typed.

Properties

The definitions we gave:

$$\begin{aligned} \texttt{even} &= \lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\setminus\texttt{Int}\to\alpha\setminus\texttt{Int})}x\,.\,x{\in}\texttt{Int}\,?\,(x \bmod 2)=0 : x \\ \texttt{map} &= \mu m^{(\alpha\to\beta)\to[\alpha]\to[\beta]}\, f\,. \\ &\qquad \lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell{\in}\texttt{nil}\,?\,\texttt{nil} : (f(\pi_1\ell), mf(\pi_2\ell)) \end{aligned}$$

are well typed.

A yardstick for the language

- ✔ Can define both `map` and `even`
- ✔ Can *check* the types specified in the signature
- ? Can *deduce* the type of the partial application `map even`

Inference of explicit type-substitutions

Two problems:

1. **Local type-substitution inference:** Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t \,?\, e : e$$

find a sound & complete algorithm that, whenever possible, inserts sets of type-substitutions making it a well-typed term of

$$e ::= x \mid ee \mid \lambda_{[\,]}^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t \,?\, e : e \mid e[\sigma_j]_{j\in J}$$

Two problems:

1. **Local type-substitution inference:** Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

find a sound & complete algorithm that, whenever possible, inserts sets of type-substitutions making it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}$$

(and, yes, the type inferred for `map even` is as expected)

Two problems:

1. **Local type-substitution inference:** Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

find a sound & complete algorithm that, whenever possible, inserts sets of type-substitutions making it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}$$

(and, yes, the type inferred for `map even` is as expected)

2. **Type reconstruction:** Given a term

$$\lambda x.e$$

find, if possible, a set of type-substitutions $[\sigma_j]_{j\in J}$ such that

$$\lambda^{\alpha\to\beta}_{[\sigma_j]_{j\in J}} x.e$$

is well typed

Local Type-Substitution Inference

Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

Infer whether it is possible to insert sets of type-substitutions in it to make it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}$$

Local Type-Substitution Inference

Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t \,?\, e : e$$

Infer whether it is possible to insert sets of type-substitutions in it to make it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e{\in}t \,?\, e : e \mid e[\sigma_j]_{j\in J}$$

No inference for decorations of λ's

Local Type-Substitution Inference

Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

Infer whether it is possible to insert sets of type-substitutions in it to make it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}$$

No inference for decorations of λ's

The reason is purely practical:

- $\lambda^{\alpha\to\alpha}x.3$ must return a static type error

Local Type-Substitution Inference

Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

Infer whether it is possible to insert sets of type-substitutions in it to make it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}$$

No inference for decorations of λ's

The reason is purely practical:

- $\lambda^{\alpha\to\alpha} x.3$ must return a static type error
- If we infer decorations, then it can be typed: $\lambda^{\alpha\to\alpha}_{\{\mathtt{Int}/_{\alpha}\}} x.3$

The rule for applications

1. In the type system: [with explicit type-subst.]

$$\textsc{(Appl)}\ \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

The rule for applications

1. In the type system:

[with explicit type-subst.]

$$\textsc{(Appl)}\quad \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination:

[with explicit type-subst.]

$$\textsc{(Appl-algorithm)}\quad \frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

The rule for applications

1. In the type system: [with explicit type-subst.]

$$(\textsc{Appl})\quad \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination: [with explicit type-subst.]

$$(\textsc{Appl-algorithm})\quad \frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

conditions for Typeability

The rule for applications

1. In the type system: [with explicit type-subst.]

(Appl)
$$\frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination: [with explicit type-subst.]

(Appl-algorithm)
$$\frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

The rule for applications

1. In the type system: [with explicit type-subst.]

$$(\textsc{Appl})\ \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination: [with explicit type-subst.]

$$(\textsc{Appl-algorithm})\ \frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

3. Inference of type substitutions [w/o explicit type-subst.]

$$(\textsc{Appl-inference})\ \frac{\exists [\sigma_i]_{i \in I}, [\sigma'_j]_{j \in J} \quad \Gamma \vdash_{\mathcal{I}} e_1 : t \qquad \Gamma \vdash_{\mathcal{I}} e_2 : s}{\Gamma \vdash_{\mathcal{I}} e_1 e_2 : \min\{u \mid t[\sigma'_j]_{j \in J} \leq s[\sigma_i]_{i \in I} \to u\}} \quad \begin{array}{l} t[\sigma'_j]_{j \in J} \leq \mathbb{0} \to \mathbb{1} \\ s[\sigma_i]_{i \in I} \leq \mathrm{dom}(t[\sigma'_j]_{j \in J}) \end{array}$$

The rule for applications

1. In the type system:

[with explicit type-subst.]

(APPL)

$$\frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination:

[with explicit type-subst.]

(APPL-ALGORITHM)

$$\frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

3. Inference of type substitutions

[w/o explicit type-subst.]

conditions for typeability

(APPL-INFERENCE)

$$\frac{\exists [\sigma_i]_{i \in I}, [\sigma'_j]_{j \in J} \quad \Gamma \vdash_{\mathcal{I}} e_1 : t \qquad \Gamma \vdash_{\mathcal{I}} e_2 : s}{\Gamma \vdash_{\mathcal{I}} e_1 e_2 : \min\{u \mid t[\sigma'_j]_{j \in J} \leq s[\sigma_i]_{i \in I} \to u\}} \quad \begin{array}{l} t[\sigma'_j]_{j \in J} \leq \mathbb{0} \to \mathbb{1} \\ s[\sigma_i]_{i \in I} \leq \mathrm{dom}(t[\sigma'_j]_{j \in J}) \end{array}$$

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, two sets $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \text{dom}(t[\sigma'_j]_{j\in J})$$

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, two sets $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \rightarrow \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \text{dom}(t[\sigma'_j]_{j\in J})$$

This can be reduced to solving a suite of *tallying problems*:

Definition (Type tallying)

Let s and t be two types. A type-substitution σ is a solution for the *tallying* of (s, t) iff $s\sigma \leq t\sigma$.

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, two sets $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \text{dom}(t[\sigma'_j]_{j\in J})$$

This can be reduced to solving a suite of *tallying problems*:

Definition (Type tallying)

Let s and t be two types. A type-substitution σ is a solution for the *tallying* of (s,t) iff $s\sigma \leq t\sigma$.

Generally: let $C = \{(s_1 \leq t_1), ..., (s_n \leq t_n)\}$ a *constraint set*. A type-substitution σ is a solution for the *tallying* of C iff $s\sigma \leq t\sigma$ for all $(s \leq t) \in C$.

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, two sets $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \text{dom}(t[\sigma'_j]_{j\in J})$$

This can be reduced to solving a suite of *tallying problems*:

Definition (Type tallying)

Let s and t be two types. A type-substitution σ is a solution for the *tallying* of (s,t) iff $s\sigma \leq t\sigma$.

Generally: let $C = \{(s_1 \leq t_1), \ldots, (s_n \leq t_n)\}$ a *constraint set*. A type-substitution σ is a solution for the *tallying* of C iff $s\sigma \leq t\sigma$ for all $(s \leq t) \in C$.

Type tallying is decidable and a sound and complete set of solutions for every tallying problem can be effectively found in **three** simple **steps**.

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Example:

1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \quad \leadsto \quad \{(s_2 \leq \mathbb{0})\}$ or $\{(s_2 \leq s_1), (t_1 \leq t_2)\}$

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 2: Merge constraints on the same variable.

- if $\alpha \leq t_1$ and $\alpha \leq t_2$ are in C, then replace them by $\alpha \leq t_1 \wedge t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in C, then replace them by $s_1 \vee s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Example:

1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \quad \rightsquigarrow \quad \{(s_2 \leq \mathbb{0})\}$ or $\{(s_2 \leq s_1), (t_1 \leq t_2)\}$

Step 1: Decompose constraints.
Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 2: Merge constraints on the same variable.

- if $\alpha \leq t_1$ and $\alpha \leq t_2$ are in C, then replace them by $\alpha \leq t_1 \wedge t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in C, then replace them by $s_1 \vee s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Example:

1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \quad \rightsquigarrow \quad \{(s_2 \leq \mathbb{0})\}$ or $\{(s_2 \leq s_1), (t_1 \leq t_2)\}$
2. $\{(\texttt{Int} \leq \alpha), (\texttt{Bool} \leq \alpha)\} \quad \rightsquigarrow \quad \{(\texttt{Int} \vee \texttt{Bool} \leq \alpha)\}$

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 2: Merge constraints on the same variable.

- if $\alpha \leq t_1$ and $\alpha \leq t_2$ are in C, then replace them by $\alpha \leq t_1 \wedge t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in C, then replace them by $s_1 \vee s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Step 3: Transform into a set of equations.

After Step 2 we have constraint-sets of the form
$\{s_i \leq \alpha_i \leq t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \vee \beta) \wedge t$ with β fresh.
2. substitute $(s \vee \beta) \wedge t$ for all α in the other constraints of C
3. repeat with another constraint

Example:

1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \rightsquigarrow \{(s_2 \leq \mathbb{0})\}$ or $\{(s_2 \leq s_1), (t_1 \leq t_2)\}$
2. $\{(\texttt{Int} \leq \alpha), (\texttt{Bool} \leq \alpha)\} \rightsquigarrow \{(\texttt{Int} \vee \texttt{Bool} \leq \alpha)\}$

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 2: Merge constraints on the same variable.

- if $\alpha{\leq}t_1$ and $\alpha{\leq}t_2$ are in C, then replace them by $\alpha \leq t_1{\wedge}t_2$;
- if $s_1{\leq}\alpha$ and $s_2{\leq}\alpha$ are in C, then replace them by $s_1{\vee}s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Step 3: Transform into a set of equations.

After Step 2 we have constraint-sets of the form

$\{s_i{\leq}\alpha_i{\leq}t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s{\vee}\beta){\wedge}t$ with β fresh.
2. substitute $(s{\vee}\beta){\wedge}t$ for all α in the other constraints of C
3. repeat with another constraint

Example:

1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \quad \rightsquigarrow \quad \{(s_2 \leq \mathbb{0})\}$ or $\{(s_2 \leq s_1), (t_1 \leq t_2)\}$
2. $\{(\mathtt{Int} \leq \alpha), (\mathtt{Bool} \leq \alpha)\} \quad \rightsquigarrow \quad \{(\mathtt{Int} \vee \mathtt{Bool} \leq \alpha)\}$
3. $\{(\mathtt{Int} \leq \alpha_1 \leq \mathsf{Real}), (\alpha_2 \leq \alpha_1{\wedge}\mathtt{Int})\}$

$$\rightsquigarrow \quad \{\alpha_1 = (\mathtt{Int}{\vee}\beta){\wedge}\mathsf{Real}), (\alpha_2 = \mathtt{Int})\}$$

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 2: Merge constraints on the same variable.

- if $\alpha \leq t_1$ and $\alpha \leq t_2$ are in C, then replace them by $\alpha \leq t_1 \wedge t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in C, then replace them by $s_1 \vee s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Step 3: Transform into a set of equations.

After Step 2 we have constraint-sets of the form $\{s_i \leq \alpha_i \leq t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \vee \beta) \wedge t$ with β fresh.
2. substitute $(s \vee \beta) \wedge t$ for all α in the other constraints of C
3. repeat with another constraint

At the end we have a sets of equations $\{\alpha_i = u_i \mid i \in [1..n]\}$ that (with some care) are *contractive*. By Courcelle there exists a solution, *ie*, a substitution for $\alpha_1, \ldots, \alpha_n$ into (possibly recursive regular) types $t_1, \ldots, t_n$ (in which the fresh β's are free variables).

Example: `map even`

Start with the following tallying problem:

$$(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq s \to \gamma$$

where $s = (\mathtt{Int} \to \mathtt{Bool}) \wedge (\alpha \setminus \mathtt{Int} \to \alpha \setminus \mathtt{Int})$ is the type of `even`

Example: map even

Start with the following tallying problem:

$$(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq s \to \gamma$$

where $s = (\texttt{Int} \to \texttt{Bool}) \wedge (\alpha \setminus \texttt{Int} \to \alpha \setminus \texttt{Int})$ is the type of even

- The algorithm generates 9 constraint-sets: one is unsatisfiable ($s \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathbb{0}\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \texttt{Int}\ ,\ \texttt{Bool} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \setminus \texttt{Int}\ ,\ \alpha \setminus \texttt{Int} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \vee \texttt{Int}\ ,\ (\alpha \setminus \texttt{Int}) \vee \texttt{Bool} \leq \beta_1\}$;

Example: map even

Start with the following tallying problem:

$$(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq s \to \gamma$$

where $s = (\texttt{Int} \to \texttt{Bool}) \wedge (\alpha \backslash \texttt{Int} \to \alpha \backslash \texttt{Int})$ is the type of even

- The algorithm generates 9 constraint-sets: one is unsatisfiable ($s \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathbb{0}\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \texttt{Int}\ ,\ \texttt{Bool} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \backslash \texttt{Int}\ ,\ \alpha \backslash \texttt{Int} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \vee \texttt{Int}\ ,\ (\alpha \backslash \texttt{Int}) \vee \texttt{Bool} \leq \beta_1\}$;
- Four solutions for γ:
  $\{\gamma = [\,] \to [\,]\}$,
  $\{\gamma = [\texttt{Int}] \to [\texttt{Bool}]\}$,
  $\{\gamma = [\alpha \backslash \texttt{Int}] \to [\alpha \backslash \texttt{Int}]\}$,
  $\{\gamma = [\alpha \vee \texttt{Int}] \to [(\alpha \backslash \texttt{Int}) \vee \texttt{Bool}]\}$.

Example: map even

Start with the following tallying problem:

$$(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq s \to \gamma$$

where $s = (\texttt{Int} \to \texttt{Bool}) \wedge (\alpha \setminus \texttt{Int} \to \alpha \setminus \texttt{Int})$ is the type of even

- The algorithm generates 9 constraint-sets: one is unsatisfiable ($s \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathbb{0}\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \texttt{Int}\ ,\ \texttt{Bool} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \setminus \texttt{Int}\ ,\ \alpha \setminus \texttt{Int} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \vee \texttt{Int}\ ,\ (\alpha \setminus \texttt{Int}) \vee \texttt{Bool} \leq \beta_1\}$;
- Four solutions for γ:
  $\{\gamma = [\,] \to [\,]\}$,
  $\{\gamma = [\texttt{Int}] \to [\texttt{Bool}]\}$,
  $\{\gamma = [\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]\}$,
  $\{\gamma = [\alpha \vee \texttt{Int}] \to [(\alpha \setminus \texttt{Int}) \vee \texttt{Bool}]\}$.
- The last two are minimal and we take their intersection:
  $\{\gamma = ([\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]) \wedge ([\alpha \vee \texttt{Int}] \to [(\alpha \setminus \texttt{Int}) \vee \texttt{Bool}])\}$

On completeness and decidability

The algorithm produces a set of solutions that is **sound** (it finds only correct solutions) and **complete** (any other solution can be derived from them).

On completeness and decidability

The algorithm produces a set of solutions that is **sound** (it finds only correct solutions) and **complete** (any other solution can be derived from them).

Decidability: The algorithm is a semi-decision procedure. We conjecture decidability (N.B.: the problem is unrelated to type-reconstruction for intersection types since we have *recursive types*).

On completeness and decidability

The algorithm produces a set of solutions that is **sound** (it finds only correct solutions) and **complete** (any other solution can be derived from them).

Decidability: The algorithm is a semi-decision procedure. We conjecture decidability (N.B.: the problem is unrelated to type-reconstruction for intersection types since we have *recursive types*).

Completeness: For every solution of the inference problem, our algorithm finds an equivalent or more general solution. However, this solution is not necessary the first solution found.

In a dully execution of the algorithm on `map even` the good solution is the second one.

On completeness and decidability

The algorithm produces a set of solutions that is **sound** (it finds only correct solutions) and **complete** (any other solution can be derived from them).

Decidability: The algorithm is a semi-decision procedure. We conjecture decidability (N.B.: the problem is unrelated to type-reconstruction for intersection types since we have *recursive types*).

Completeness: For every solution of the inference problem, our algorithm finds an equivalent or more general solution. However, this solution is not necessary the first solution found.

In a dully execution of the algorithm on `map even` the good solution is the second one.

Principality: This raises the problem of the existence of principal types: may an infinite sequence of increasingly general solutions exist?

Type reconstruction

- Solve *sets* of contraint-sets by the tallying algorithm:

$$\frac{}{\Gamma \vdash_{\mathcal{R}} x : \Gamma(x) \leadsto \{\varnothing\}} \qquad \frac{\Gamma, x : \alpha \vdash_{\mathcal{R}} e : t \leadsto \mathcal{S}}{\Gamma \vdash_{\mathcal{R}} \lambda x.e : \alpha \to \beta \leadsto \mathcal{S} \sqcap \{\{(t \leq \beta)\}\}}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} e_1 : t_1 \leadsto \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} e_2 : t_2 \leadsto \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} e_1 e_2 : \alpha \leadsto \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1 \leq t_2 \to \alpha)\}\}} \quad + \quad \text{rule for typecase}$$

Type reconstruction

- Solve *sets* of contraint-sets by the tallying algorithm:

$$\frac{}{\Gamma \vdash_{\mathcal{R}} x : \Gamma(x) \leadsto \{\varnothing\}} \qquad \frac{\Gamma, x : \alpha \vdash_{\mathcal{R}} e : t \leadsto \mathcal{S}}{\Gamma \vdash_{\mathcal{R}} \lambda x.e : \alpha \to \beta \leadsto \mathcal{S} \sqcap \{\{(t \leq \beta)\}\}}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} e_1 : t_1 \leadsto \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} e_2 : t_2 \leadsto \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} e_1 e_2 : \alpha \leadsto \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1 \leq t_2 \to \alpha)\}\}} \quad + \quad \text{rule for typecase}$$

- Sound. it's a variant: fix interfaces and infer decorations

$$\lambda^{\alpha\to\beta}_{[?]} x.e$$

  Not complete: reconstruction is undecidable

Type reconstruction

- Solve *sets* of contraint-sets by the tallying algorithm:

$$\frac{}{\Gamma \vdash_{\mathcal{R}} x : \Gamma(x) \rightsquigarrow \{\varnothing\}} \qquad \frac{\Gamma, x : \alpha \vdash_{\mathcal{R}} e : t \rightsquigarrow \mathcal{S}}{\Gamma \vdash_{\mathcal{R}} \lambda x.e : \alpha \to \beta \rightsquigarrow \mathcal{S} \sqcap \{\{(t \leq \beta)\}\}}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} e_1 : t_1 \rightsquigarrow \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} e_2 : t_2 \rightsquigarrow \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} e_1 e_2 : \alpha \rightsquigarrow \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1 \leq t_2 \to \alpha)\}\}} \quad + \quad \text{rule for typecase}$$

- Sound. it's a variant: fix interfaces and infer decorations

$$\lambda^{\alpha\to\beta}_{[?]} x.e$$

  Not complete: reconstruction is undecidable

- It types more than ML

$$\lambda x.xx : \mu X.(\alpha \wedge (X \to \beta)) \to \beta \qquad (\leq \alpha \wedge (\alpha \to \beta)) \to \beta)$$

  and for functions typable in ML, it deduces a type at least as good:

$$map : ((\alpha \to \beta) \to [\alpha] \to [\beta]) \wedge ((\mathbb{0} \to \mathbb{1}) \to [] \to [])$$

Efficient evaluation

Monomorphic language

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \lambda^t x.e \mid ee \mid e\in t\,?\,e : e \\
v & ::= & c \mid \langle \lambda^t x.e, \mathcal{E} \rangle
\end{array}
$$

Monomorphic language

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \lambda^t x.e \mid ee \mid e \in t \, ? \, e : e \\
v & ::= & c \mid \langle \lambda^t x.e, \mathcal{E} \rangle
\end{array}
$$

$$
(\textsc{Closure}) \; \frac{}{\mathcal{E} \vdash_{\mathsf{m}} \lambda^t x.e \Downarrow \langle \lambda^t x.e, \mathcal{E} \rangle}
$$

$$
(\textsc{Apply}) \; \frac{\mathcal{E} \vdash_{\mathsf{m}} e_1 \Downarrow \langle \lambda^t x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash_{\mathsf{m}} e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash_{\mathsf{m}} e \Downarrow v}{\mathcal{E} \vdash_{\mathsf{m}} e_1 e_2 \Downarrow v}
$$

Monomorphic language

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}x.e \mid ee \mid e \in t \,?\, e : e \\
v & ::= & c \mid \langle \lambda^{t}x.e, \mathcal{E} \rangle
\end{array}
$$

save the environment

$$
(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash_{\mathrm{m}} \lambda^{t}x.e \Downarrow \langle \lambda^{t}x.e, \mathcal{E} \rangle}
$$

$$
(\textsc{Apply})\ \frac{\mathcal{E} \vdash_{\mathrm{m}} e_1 \Downarrow \langle \lambda^{t}x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash_{\mathrm{m}} e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash_{\mathrm{m}} e \Downarrow v}{\mathcal{E} \vdash_{\mathrm{m}} e_1 e_2 \Downarrow v}
$$

Monomorphic language

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \lambda^t x.e \mid ee \mid e \in t\, ?\, e : e \\
v & ::= & c \mid \langle \lambda^t x.e, \mathcal{E} \rangle
\end{array}
$$

save the environment

$$(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash_{\mathsf{m}} \lambda^t x.e \Downarrow \langle \lambda^t x.e, \mathcal{E} \rangle}$$

$$(\textsc{Apply})\ \frac{\mathcal{E} \vdash_{\mathsf{m}} e_1 \Downarrow \langle \lambda^t x.e, \mathcal{E}' \rangle \quad \mathcal{E} \vdash_{\mathsf{m}} e_2 \Downarrow v_0 \quad \mathcal{E}', x \mapsto v_0 \vdash_{\mathsf{m}} e \Downarrow v}{\mathcal{E} \vdash_{\mathsf{m}} e_1 e_2 \Downarrow v}$$

restore the environment

Monomorphic language

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^t x.e \mid ee \mid e \in t\, ?\, e : e \\
v & ::= & c \mid \langle \lambda^t x.e, \mathcal{E} \rangle
\end{array}
$$

$$
(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash_{\mathsf{m}} \lambda^t x.e \Downarrow \langle \lambda^t x.e, \mathcal{E} \rangle}
$$

$$
(\textsc{Apply})\ \frac{\mathcal{E} \vdash_{\mathsf{m}} e_1 \Downarrow \langle \lambda^t x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash_{\mathsf{m}} e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash_{\mathsf{m}} e \Downarrow v}{\mathcal{E} \vdash_{\mathsf{m}} e_1 e_2 \Downarrow v}
$$

$$
(\textsc{Typecase True})\ \frac{\mathcal{E} \vdash_{\mathsf{m}} e_1 \Downarrow v_0 \qquad v_0 \in_{\mathsf{m}} t \qquad \mathcal{E} \vdash_{\mathsf{m}} e_2 \Downarrow v}{\mathcal{E} \vdash_{\mathsf{m}} e_1 \in t\, ?\, e_2 : e_3 \Downarrow v}
$$

$$
(\textsc{Typecase False})\ \frac{\mathcal{E} \vdash_{\mathsf{m}} e_1 \Downarrow v_0 \qquad v_0 \notin_{\mathsf{m}} t \qquad \mathcal{E} \vdash_{\mathsf{m}} e_3 \Downarrow v}{\mathcal{E} \vdash_{\mathsf{m}} e_1 \in t\, ?\, e_2 : e_3 \Downarrow v}
$$

$$
\begin{array}{rcl}
c \in_{\mathsf{m}} t & \stackrel{\text{def}}{=} & \{c\} \leq t \\
\langle \lambda^s x.e, \mathcal{E} \rangle \in_{\mathsf{m}} t & \stackrel{\text{def}}{=} & s \leq t
\end{array}
$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i \in I}$)

$$e \quad ::= \quad c \mid x \mid \lambda^{t}_{\sigma_I} x.e \mid ee \mid e \in t \,?\, e : e \mid e\sigma_I$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e \quad ::= \quad c \mid x \mid \lambda^{t}_{\sigma_I} x.e \mid ee \mid e \in t\ ?\ e : e \mid e\sigma_I$$
$$v \quad ::= \quad c \mid \langle \lambda^{t}_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e \quad ::= \quad c \mid x \mid \lambda^t_{\sigma_I} x.e \mid ee \mid e \in t \,?\, e : e \mid e\sigma_I$$
$$v \quad ::= \quad c \mid \langle \lambda^t_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle$$

$$(\textsc{Closure})\ \frac{}{\sigma_I ; \mathcal{E} \vdash_{\mathsf{p}} \lambda^t_{\sigma_J} x.e \Downarrow \langle \lambda^t_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle}$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e ::= c \mid x \mid \lambda^t_{\sigma_I} x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I$$
$$v ::= c \mid \langle \lambda^t_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle$$

save the environment

$$(\textsc{Closure})\ \frac{}{\sigma_I; \mathcal{E} \vdash_{\mathsf{p}} \lambda^t_{\sigma_J} x.e \Downarrow \langle \lambda^t_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle}$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e ::= c \mid x \mid \lambda^t_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I$$
$$v ::= c \mid \langle\lambda^t_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle$$

save the environment

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathsf{p}}\lambda^t_{\sigma_J}x.e \Downarrow \langle\lambda^t_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}$$

save current type-substitutions

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I} x.e \mid ee \mid e \in t \,?\, e : e \mid e\sigma_I \\
v & ::= & c \mid \langle \lambda^{t}_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle
\end{array}
$$

$$
(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E} \vdash_{\mathsf{p}} \lambda^{t}_{\sigma_J} x.e \Downarrow \langle \lambda^{t}_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle}
\qquad
(\textsc{Instance})\ \frac{\sigma_I \circ \sigma_J;\mathcal{E} \vdash_{\mathsf{p}} e \Downarrow v}{\sigma_I;\mathcal{E} \vdash_{\mathsf{p}} e\sigma_J \Downarrow v}
$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I \\
v & ::= & c \mid \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle
\end{array}
$$

$$
(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} \lambda^{t}_{\sigma_J}x.e \Downarrow \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle}
\qquad
(\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e\sigma_J\Downarrow v}
$$

(Apply)

$$
\frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1e_2\Downarrow v}
$$

where $\sigma_J = \sigma_H\circ\sigma_K$ and $P = \{j\in J \mid \exists \ell\in L : v_0\in_{\mathrm{p}} s_\ell\sigma_j\}$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e ::= c \mid x \mid \lambda^t_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I$$
$$v ::= c \mid \langle\lambda^t_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\rm p}\lambda^t_{\sigma_J}x.e \Downarrow \langle\lambda^t_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}
\qquad
(\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\rm p} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\rm p} e\sigma_J\Downarrow v}$$

restore the environment

(Apply)

$$\frac{\sigma_I;\mathcal{E}\vdash_{\rm p} e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_{\rm p} e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\rm p} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\rm p} e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J\mid \exists\ell\in L: v_0\in_{\rm p} s_\ell\sigma_j\}$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e \;::=\; c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I$$
$$v \;::=\; c \mid \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm p}\lambda^{t}_{\sigma_J}x.e \Downarrow \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle} \qquad (\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm p} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm p} e\sigma_J\Downarrow v}$$

restore the environment

$$(\textsc{Apply})\ \frac{\sigma_I;\mathcal{E}\vdash_{\mathrm p} e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_{\mathrm p} e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm p} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm p} e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J\mid \exists \ell\in L: v_0\in_{\mathrm p} s_\ell\sigma_j\}$

restore the type substitutions

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e ::= c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e : e \mid e\sigma_I$$
$$v ::= c \mid \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} \lambda^{t}_{\sigma_J}x.e \Downarrow \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}
\qquad
(\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e\sigma_J\Downarrow v}$$

$$(\textsc{Apply})\ \frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1e_2\Downarrow v}$$

where $\sigma_J = \sigma_H\circ\sigma_K$ and $P = \{j\in J \mid \exists\ell\in L : v_0\in_{\mathrm{p}} s_\ell\sigma_j\}$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e ::= c \mid x \mid \lambda^t_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e : e \mid e\sigma_I$$
$$v ::= c \mid \langle\lambda^t_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}\lambda^t_{\sigma_J}x.e \Downarrow \langle\lambda^t_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle} \qquad (\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e\sigma_J\Downarrow v}$$

(Apply)

$$\frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1e_2\Downarrow v}$$

where $\sigma_J = \sigma_H\circ\sigma_K$ and $P = \{j\in J \mid \exists\ell\in L : v_0 \in_{\mathrm{p}} s_\ell\sigma_j\}$

Problem:

At every application compute σ_P:

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e : e \mid e\sigma_I \\
v & ::= & c \mid \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle
\end{array}
$$

$$
(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} \lambda^{t}_{\sigma_J}x.e \Downarrow \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle}
\qquad
(\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e\sigma_J\Downarrow v}
$$

(Apply)

$$
\frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1\Downarrow \langle \lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e, \mathcal{E}', \sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1e_2\Downarrow v}
$$

where $\sigma_J = \sigma_H\circ\sigma_K$ and $P = \{j\in J \mid \exists \ell\in L : v_0 \in_{\mathrm{p}} s_\ell\sigma_j\}$

Problem:

At every application compute σ_P:

1. **compose** of two sets of type-substitution

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e ::= c \mid x \mid \lambda^t_{\sigma_I} x.e \mid ee \mid e \in t\ ?\ e : e \mid e\sigma_I$$
$$v ::= c \mid \langle \lambda^t_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle$$

$$(\text{Closure}) \ \frac{}{\sigma_I; \mathcal{E} \vdash_p \lambda^t_{\sigma_J} x.e \Downarrow \langle \lambda^t_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle} \qquad (\text{Instance}) \ \frac{\sigma_I \circ \sigma_J; \mathcal{E} \vdash_p e \Downarrow v}{\sigma_I; \mathcal{E} \vdash_p e\sigma_J \Downarrow v}$$

(Apply)

$$\frac{\sigma_I; \mathcal{E} \vdash_p e_1 \Downarrow \langle \lambda^{\wedge_{\ell\in L} s_\ell \to t_\ell}_{\sigma_K} x.e, \mathcal{E}', \sigma_H \rangle \quad \sigma_I; \mathcal{E} \vdash_p e_2 \Downarrow v_0 \quad \sigma_P; \mathcal{E}', x \mapsto v_0 \vdash_p e \Downarrow v}{\sigma_I; \mathcal{E} \vdash_p e_1 e_2 \Downarrow v}$$

where $\sigma_J = \sigma_H \circ \sigma_K$ and $P = \{j \in J \mid \exists \ell \in L : v_0 \in_p s_\ell \sigma_j\}$

Problem:

At every application compute σ_P:

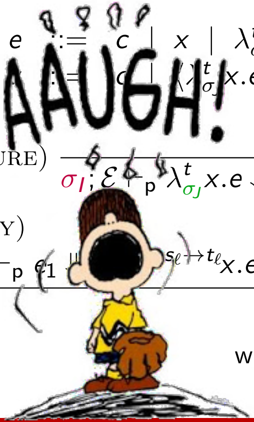
1. **compose** of two sets of type-substitution
2. **select** the substitutions compatible with the argument v_0

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e ::= c \mid x \mid \lambda^t_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I$$

$$v ::= c \mid \langle\lambda^t_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle$$



(Closure) $$\frac{}{\sigma_I;\mathcal{E}\vdash_p \lambda^t_{\sigma_J}x.e \Downarrow \langle\lambda^t_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}$$

(Instance) $$\frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_p e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_p e\sigma_J\Downarrow v}$$

(Apply)
$$\frac{\sigma_I;\mathcal{E}\vdash_p e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_J}x.e,\mathcal{E}',\sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_p e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_p e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_p e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J\mid \exists\ell\in L: v_0\in_p s_\ell\sigma_j\}$

Problem:

At every application compute σ_P:

1. **compose** of two sets of type-substitution
2. **select** the substitutions compatible with the argument v_0

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e ::= c \mid x \mid \lambda^t_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e : e \mid e\sigma_I$$
$$v ::= c \mid \langle\lambda^t_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathsf{p}}\lambda^t_{\sigma_J}x.e \Downarrow \langle\lambda^t_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle} \qquad (\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathsf{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathsf{p}} e\sigma_J\Downarrow v}$$

(Apply)

$$\frac{\sigma_I;\mathcal{E}\vdash_{\mathsf{p}} e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_{\mathsf{p}} e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathsf{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathsf{p}} e_1e_2\Downarrow v}$$

where $\sigma_J = \sigma_H\circ\sigma_K$ and $P = \{j\in J \mid \exists\ell\in L : v_0\in_{\mathsf{p}} s_\ell\sigma_j\}$

Solution:

Compute compositions and selections lazily.

Intermediate language as compilation target

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t} x.e \mid ee \mid e \in t \,?\, e : e \\
v & ::= & c \mid \langle \lambda^{t} x.e, \mathcal{E} \rangle
\end{array}
$$

$$
(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^{t} x.e \Downarrow \langle \lambda^{t} x.e, \mathcal{E} \rangle}
$$

$$
(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^{t} x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}
$$

$$
(\textsc{Typecase True})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t \,?\, e_2 : e_3 \Downarrow v}
\qquad
(\textsc{Typecase False})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t \,?\, e_2 : e_3 \Downarrow v}
$$

$$
\begin{array}{rcl}
c \in t & \stackrel{\text{def}}{=} & \{c\} \leq t \\
\langle \lambda^{s} x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{=} & s \leq t
\end{array}
$$

Intermediate language as compilation target

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^t_{\Sigma} x.e \mid ee \mid e \in t \,?\, e : e \\
v & ::= & c \mid \langle \lambda^t_{\Sigma} x.e, \mathcal{E} \rangle \\
\Sigma & ::= & \sigma_I \mid \mathtt{comp}(\Sigma, \Sigma') \mid \mathtt{sel}(x, t, \Sigma)
\end{array}
$$

symbolic substitutions

$$
(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^t x.e \Downarrow \langle \lambda^t x.e, \mathcal{E} \rangle}
$$

$$
(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^t x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}
$$

$$
(\textsc{Typecase True})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t \,?\, e_2 : e_3 \Downarrow v}
\qquad
(\textsc{Typecase False})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t \,?\, e_2 : e_3 \Downarrow v}
$$

$$
\begin{array}{rcl}
c \in t & \stackrel{\text{def}}{=} & \{c\} \leq t \\
\langle \lambda^s x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{=} & s \leq t
\end{array}
$$

Intermediate language as compilation target

$$e ::= c \mid x \mid \lambda^t_\Sigma x.e \mid ee \mid e \in t\,?\,e : e$$
$$v ::= c \mid \langle \lambda^t_\Sigma x.e, \mathcal{E} \rangle$$
$$\Sigma ::= \sigma_I \mid \mathtt{comp}(\Sigma, \Sigma') \mid \mathtt{sel}(x, t, \Sigma)$$

symbolic substitutions

$$(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^t_\Sigma x.e \Downarrow \langle \lambda^t_\Sigma x.e, \mathcal{E} \rangle}$$

$$(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^t_\Sigma x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}$$

(Typecase True)
$$\frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

(Typecase False)
$$\frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$c \in t \stackrel{\text{def}}{=} \{c\} \leq t$$
$$\langle \lambda^s x.e, \mathcal{E} \rangle \in t \stackrel{\text{def}}{=} s \leq t$$

Intermediate language as compilation target

$$
\begin{array}{lll}
e & ::= & c \mid x \mid \lambda^t_\Sigma x.e \mid ee \mid e \in t \,?\, e : e \\
v & ::= & c \mid \langle \lambda^t_\Sigma x.e, \mathcal{E} \rangle \\
\Sigma & ::= & \sigma_I \mid \mathtt{comp}(\Sigma, \Sigma') \mid \mathtt{sel}(x, t, \Sigma)
\end{array}
$$

symbolic substitutions

$$
(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^t_\Sigma x.e \Downarrow \langle \lambda^t_\Sigma x.e, \mathcal{E} \rangle}
$$

$$
(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^t_\Sigma x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}
$$

$$
(\textsc{Typecase True})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t \,?\, e_2 : e_3 \Downarrow v}
\qquad
(\textsc{Typecase False})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t \,?\, e_2 : e_3 \Downarrow v}
$$

$$
\begin{array}{rcl}
c \in t & \stackrel{\text{def}}{=} & \{c\} \leq t \\
\langle \lambda^s x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{=} & s \leq t
\end{array}
$$

Intermediate language as compilation target

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \lambda^t_\Sigma x.e \mid ee \mid e \in t\,?\,e : e \\
v & ::= & c \mid \langle \lambda^t_\Sigma x.e, \mathcal{E} \rangle \\
\Sigma & ::= & \sigma_I \mid \texttt{comp}(\Sigma, \Sigma') \mid \texttt{sel}(x, t, \Sigma)
\end{array}
$$

symbolic substitutions

$$
(\textsc{Closure}) \quad \frac{}{\mathcal{E} \vdash \lambda^t_\Sigma x.e \Downarrow \langle \lambda^t_\Sigma x.e, \mathcal{E} \rangle}
$$

$$
(\textsc{Apply}) \quad \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^t_\Sigma x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}
$$

$$
(\textsc{Typecase True}) \quad \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}
$$

$$
(\textsc{Typecase False}) \quad \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}
$$

watch here !

$$
\begin{array}{rcl}
c \in t & \stackrel{\text{def}}{=} & \{c\} \leq t \\
\langle \lambda^s x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{=} & s \leq t
\end{array}
$$

Intermediate language as compilation target

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}_{\Sigma}x.e \mid ee \mid e \in t\,?\,e : e \\
v & ::= & c \mid \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E} \rangle \\
\Sigma & ::= & \sigma_I \mid \texttt{comp}(\Sigma, \Sigma') \mid \texttt{sel}(x, t, \Sigma)
\end{array}
$$

symbolic substitutions

$$(\text{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^{t}_{\Sigma}x.e \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E} \rangle}$$

$$(\text{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}$$

$$(\text{Typecase True})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$(\text{Typecase False})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$
\begin{array}{rcl}
c \in t & \stackrel{\text{def}}{=} & \{c\} \leq t \\
\langle \lambda^{s}_{\Sigma}x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{=} & s(\text{eval}(\mathcal{E}, \Sigma)) \leq t
\end{array}
$$

Intermediate language as compilation target

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}_{\Sigma}x.e \mid ee \mid e\in t\,?\,e : e \\
v & ::= & c \mid \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}\rangle \\
\Sigma & ::= & \sigma_I \mid \mathtt{comp}(\Sigma,\Sigma') \mid \mathtt{sel}(x,t,\Sigma)
\end{array}
$$

symbolic substitutions

$$(\textsc{Closure})\ \frac{}{\mathcal{E}\vdash \lambda^{t}_{\Sigma}x.e \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}\rangle}$$

$$(\textsc{Apply})\ \frac{\mathcal{E}\vdash e_1 \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}'\rangle \qquad \mathcal{E}\vdash e_2\Downarrow v_0 \qquad \mathcal{E}', x\mapsto v_0 \vdash e \Downarrow v}{\mathcal{E}\vdash e_1 e_2 \Downarrow v}$$

$$(\textsc{Typecase True})\ \frac{\mathcal{E}\vdash e_1\Downarrow v_0 \qquad v_0\in t \qquad \mathcal{E}\vdash e_2\Downarrow v}{\mathcal{E}\vdash e_1\in t\,?\,e_2 : e_3 \Downarrow v}$$

$$(\textsc{Typecase False})\ \frac{\mathcal{E}\vdash e_1\Downarrow v_0 \qquad v_0\notin t \qquad \mathcal{E}\vdash e_3\Downarrow v}{\mathcal{E}\vdash e_1\in t\,?\,e_2 : e_3 \Downarrow v}$$

The only difference!

$$c \in t \overset{\text{def}}{=} \{c\} \leq t$$

$$\langle \lambda^{s}_{\Sigma}x.e, \mathcal{E}\rangle \in t \overset{\text{def}}{=} s(\mathsf{eval}(\mathcal{E},\Sigma)) \leq t$$

Compilation

1. Compile into the intermediate language

$$
\begin{array}{rcl}
[\![x]\!]_\Sigma & = & x \\
[\![\lambda^t_{\sigma_I} x.e]\!]_\Sigma & = & \lambda^t_{\mathtt{comp}(\Sigma,\sigma_I)} x.[\![e]\!]_{\mathtt{sel}(x,t,\mathtt{comp}(\Sigma,\sigma_I))} \\
[\![e_1 e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma \\
[\![e\sigma_I]\!]_\Sigma & = & [\![e]\!]_{\mathtt{comp}(\Sigma,\sigma_I)} \\
[\![e_1 \in t \,?\, e_2 : e_3]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \in t \,?\, [\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma
\end{array}
$$

Compilation

1. Compile into the intermediate language

$$
\begin{array}{rcl}
[\![x]\!]_\Sigma & = & x \\
[\![\lambda^t_{\sigma_I} x.e]\!]_\Sigma & = & \lambda^t_{\mathtt{comp}(\Sigma,\sigma_I)} x.[\![e]\!]_{\mathtt{sel}(x,t,\mathtt{comp}(\Sigma,\sigma_I))} \\
[\![e_1 e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma \\
[\![e\sigma_I]\!]_\Sigma & = & [\![e]\!]_{\mathtt{comp}(\Sigma,\sigma_I)} \\
[\![e_1 \in t \,?\, e_2 : e_3]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \in t \,?\, [\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma
\end{array}
$$

2. For $\langle \lambda^s_\Sigma x.e, \mathcal{E}\rangle \in t \stackrel{\text{def}}{=} s(\text{eval}(\mathcal{E},\Sigma)) \leq t$ we have $s(\text{eval}(\mathcal{E},\Sigma)) \neq s$ *only if* $\lambda^s_\Sigma x.e$ results from the partial application of a polymorphic function (*ie*, in s there occur free variables bound in the context).

Compilation

1. Compile into the intermediate language

$$
\begin{array}{rcl}
[\![x]\!]_\Sigma & = & x \\
[\![\lambda^t_{\sigma_I} x.e]\!]_\Sigma & = & \lambda^t_{\texttt{comp}(\Sigma,\sigma_I)} x.[\![e]\!]_{\texttt{sel}(x,t,\texttt{comp}(\Sigma,\sigma_I))} \\
[\![e_1 e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma \\
[\![e\sigma_I]\!]_\Sigma & = & [\![e]\!]_{\texttt{comp}(\Sigma,\sigma_I)} \\
[\![e_1 \in t\ ?\ e_2 : e_3]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \in t\ ?\ [\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma
\end{array}
$$

2. For $\langle \lambda^s_\Sigma x.e, \mathcal{E} \rangle \in t \stackrel{\text{def}}{=} s(\text{eval}(\mathcal{E}, \Sigma)) \leq t$ we have $s(\text{eval}(\mathcal{E}, \Sigma)) \neq s$ *only if* $\lambda^s_\Sigma x.e$ results from the partial application of a polymorphic function (*ie*, in s there occur free variables bound in the context).

> Execution *may* be slowed *only* when testing the type of the result of a partial application of a polymorphic function.

3. Compilation can flag the functions that may require to compute eval:

$$[\![\lambda^{t}_{[\!]}x.e]\!]_{\Sigma} \quad = \quad \begin{cases} \lambda^{t}_{\Sigma}x.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma)} & \text{if } \mathsf{var}(t) \cap \mathsf{dom}(\Sigma) = \varnothing \\ \hat{\boldsymbol{\lambda}}^{t}_{\Sigma}x.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma)} & \text{otherwise} \end{cases}$$

and then we evaluate the symbolic substitutions only for marked functions:

$$\begin{array}{lcl} \langle\lambda^{s}_{\Sigma}x.e,\mathcal{E}\rangle \in t & \stackrel{\text{def}}{\Longleftrightarrow} & s \leq t \\ \langle\hat{\boldsymbol{\lambda}}^{s}_{\Sigma}x.e,\mathcal{E}\rangle \in t & \stackrel{\text{def}}{\Longleftrightarrow} & s(\mathsf{eval}(\mathcal{E},\Sigma)) \leq t \end{array}$$

3. Compilation can flag the functions that may require to compute eval:

$$[\![\lambda^{t}_{\mathbb{I}} x.e]\!]_{\Sigma} \quad = \quad \begin{cases} \lambda^{t}_{\Sigma} x.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma)} & \text{if } \mathsf{var}(t) \cap \mathsf{dom}(\Sigma) = \varnothing \\ \hat{\boldsymbol{\lambda}}^{t}_{\Sigma} x.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma)} & \text{otherwise} \end{cases}$$

and then we evaluate the symbolic substitutions only for marked functions:

$$\begin{array}{lcl} \langle \lambda^{s}_{\Sigma} x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{\Longleftrightarrow} & s \leq t \\ \langle \hat{\boldsymbol{\lambda}}^{s}_{\Sigma} x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{\Longleftrightarrow} & s(\mathsf{eval}(\mathcal{E}, \Sigma)) \leq t \end{array}$$

4. This holds also with products (used to encode lists records and XML), whose testing accounts for most of the execution time.

3. Compilation can flag the functions that may require to compute eval:

$$[\![\lambda^{t}_{\mathbb{i}} x.e]\!]_\Sigma \quad = \quad \begin{cases} \lambda^{t}_{\Sigma} x.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma)} & \text{if } \mathsf{var}(t) \cap \mathsf{dom}(\Sigma) = \varnothing \\ \hat{\boldsymbol{\lambda}}^{t}_{\Sigma} x.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma)} & \text{otherwise} \end{cases}$$

and then we evaluate the symbolic substitutions only for marked functions:

$$\begin{array}{lcl} \langle \lambda^{s}_{\Sigma} x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{\Longleftrightarrow} & s \leq t \\ \langle \hat{\boldsymbol{\lambda}}^{s}_{\Sigma} x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{\Longleftrightarrow} & s(\mathsf{eval}(\mathcal{E}, \Sigma)) \leq t \end{array}$$

4. This holds also with products (used to encode lists records and XML), whose testing accounts for most of the execution time.

Bottom Line

The execution is as efficient as in the monorphic case, apart from a single well identified exception

Conclusion

Theory: All the theoretical machinery necessary to design and implement a programming language is there. The practical relevance of the open theoretical issues is negligible.

Theory: All the theoretical machinery necessary to design and implement a programming language is there. The practical relevance of the open theoretical issues is negligible.

Languages: The implementation of the polymorphic extension of ℂDuce is almost done (see git); we intend to study the definition of polymorphic extensions of XQuery and to embed some of this type machinery in ML (e.g., type `balance` for red-black trees).

Theory: All the theoretical machinery necessary to design and implement a programming language is there. The practical relevance of the open theoretical issues is negligible.

Languages: The implementation of the polymorphic extension of $\mathbb{C}$Duce is almost done (see git); we intend to study the definition of polymorphic extensions of XQuery and to embed some of this type machinery in ML (e.g., type `balance` for red-black trees).

Runtime: Relabeling cannot be avoided but it is materialized only in case of partial polymorphic applications that end up in type-cases, that is, just when it is needed.

Theory: All the theoretical machinery necessary to design and implement a programming language is there. The practical relevance of the open theoretical issues is negligible.

Languages: The implementation of the polymorphic extension of ℂDuce is almost done (see git); we intend to study the definition of polymorphic extensions of XQuery and to embed some of this type machinery in ML (e.g., type `balance` for red-black trees).

Runtime: Relabeling cannot be avoided but it is materialized only in case of partial polymorphic applications that end up in type-cases, that is, just when it is needed.

Implementation: Subtyping of polymorphic types require minimal modifications to the implementation. Existing data structures (e.g., binary decision trees with lazy unions) and optimizations mostly transpose smoothly.

Theory: All the theoretical machinery necessary to design and implement a programming language is there. The practical relevance of the open theoretical issues is negligible.

Languages: The implementation of the polymorphic extension of $\mathbb{C}$Duce is almost done (see git); we intend to study the definition of polymorphic extensions of XQuery and to embed some of this type machinery in ML (e.g., type `balance` for red-black trees).

Runtime: Relabeling cannot be avoided but it is materialized only in case of partial polymorphic applications that end up in type-cases, that is, just when it is needed.

Implementation: Subtyping of polymorphic types require minimal modifications to the implementation. Existing data structures (e.g., binary decision trees with lazy unions) and optimizations mostly transpose smoothly.

Type reconstruction: Full usage needs more research, expecially about the production of human readable types and helpful error messages, but it is mature enough to use it to type local functions.

Extra slides

References

1. *Subtyping:* Set-theoretic Foundation of Parametric Polymorphism and Subtyping. **ICFP '11**
2. *Language:* Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation. **POPL '14**
3. *Language:* Polymorphic Functions with Set-Theoretic Types. Part 2: Local Type Inference and Type Reconstruction. **POPL '15**.

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \text{dom}(t[\sigma'_j]_{j\in J})$$

This can be reduced to solving a suite of *tallying problems*:

Definition (Type tallying)

Let $C = \{(s_1, t_1), ..., (s_n, t_n)\}$ a *constraint set*. A type-substitution σ is a solution for the *tallying* of C iff $s\sigma \leq t\sigma$ for all $(s, t) \in C$.

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \mathrm{dom}(t[\sigma'_j]_{j\in J})$$

This can be reduced to solving a suite of *tallying problems*:

Definition (Type tallying)

Let $C = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ a *constraint set*. A type-substitution σ is a solution for the *tallying* of C iff $s\sigma \leq t\sigma$ for all $(s, t) \in C$.

Type tallying is decidable and a sound and complete set of solutions for every tallying problem can be effectively found in three simple steps.

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form (α, t) or (t, α).

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form (α, t) or (t, α).

Step 2: Merge constraints on the same variable.

- if (α, t_1) and (α, t_2) are in C, then replace them by $(\alpha, t_1 \wedge t_2)$;
- if (s_1, α) and (s_2, α) are in C, then replace them by $(s_1 \vee s_2, \alpha)$;

Possibly decompose the new constraints generated by transitivity.

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form (α, t) or (t, α).

Step 2: Merge constraints on the same variable.

- if (α, t_1) and (α, t_2) are in C, then replace them by $(\alpha, t_1 \wedge t_2)$;
- if (s_1, α) and (s_2, α) are in C, then replace them by $(s_1 \vee s_2, \alpha)$;

Possibly decompose the new constraints generated by transitivity.

Step 3: Transform into a set of equations.

After Step 2 we have constraint-sets of the form
$\{s_i \leq \alpha_i \leq t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \vee \beta) \wedge t$ with β fresh.
2. in all other constraints in replace every α by $(s \vee \beta) \wedge t$
3. repeat with another constraint

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form (α, t) or (t, α).

Step 2: Merge constraints on the same variable.

- if (α, t_1) and (α, t_2) are in C, then replace them by $(\alpha, t_1 \wedge t_2)$;
- if (s_1, α) and (s_2, α) are in C, then replace them by $(s_1 \vee s_2, \alpha)$;

Possibly decompose the new constraints generated by transitivity.

Step 3: Transform into a set of equations.

After Step 2 we have constraint-sets of the form $\{s_i \leq \alpha_i \leq t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \vee \beta) \wedge t$ with β fresh.
2. in all other constraints in replace every α by $(s \vee \beta) \wedge t$
3. repeat with another constraint

At the end we have a sets of equations $\{\alpha_i = u_i \mid i \in [1..n]\}$ that (with some care) are *contractive*. By Courcelle there exists a solution, *ie*, a substitution for $\alpha_1, ..., \alpha_n$ into (possibly recursive regular) types $t_1, ..., t_n$ (in which the fresh β's are free variables).

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);
2. Split each substitution σ_k (for $k{\in}I{\cup}J$) in two: $\sigma_k = \rho_k \circ \sigma'_k$ where ρ_k is a renaming substitution mapping each variable of the domain of σ_k into a fresh variable:

$$\bigwedge_{i\in I}(t\rho_i)\sigma'_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J}(s\rho_j)\sigma'_j \leq \mathrm{dom}(\ \bigwedge_{i\in I}(t\rho_i)\sigma'_i);$$

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \text{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);
2. Split each substitution σ_k (for $k\in I\cup J$) in two: $\sigma_k = \rho_k \circ \sigma'_k$ where ρ_k is a renaming substitution mapping each variable of the domain of σ_k into a fresh variable:
   $(\bigwedge_{i\in I} t\rho_i)\sigma \leq \mathbb{0}\to\mathbb{1}$ and $(\bigwedge_{j\in J} s\rho_j)\sigma \leq \text{dom}((\bigwedge_{i\in I} t\rho_i)\sigma)$;

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);
2. Split each substitution σ_k (for $k\in I\cup J$) in two: $\sigma_k = \rho_k \circ \sigma'_k$ where ρ_k is a renaming substitution mapping each variable of the domain of σ_k into a fresh variable:
   $(\bigwedge_{i\in I} t\rho_i)\sigma \leq \mathbb{0}\to\mathbb{1}$ and $(\bigwedge_{j\in J} s\rho_j)\sigma \leq \mathrm{dom}((\bigwedge_{i\in I} t\rho_i)\sigma)$;
3. Solve the tallying problem for
   $$\{(t_1, \mathbb{0}\to\mathbb{1}), (t_1, t_2\to\gamma)\}$$
   with $t_1 = \bigwedge_{i\in I} t\rho_i$, $t_2 = \bigwedge_{j\in J} s\rho_j$, and γ fresh

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \;\leq\; \mathbb{0}\to\mathbb{1} \quad\text{and}\quad \bigwedge_{j\in J} s\sigma_j \;\leq\; \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);
2. Split each substitution σ_k (for $k\in I\cup J$) in two: $\sigma_k = \rho_k \circ \sigma'_k$ where ρ_k is a renaming substitution mapping each variable of the domain of σ_k into a fresh variable: $(\bigwedge_{i\in I} t\rho_i)\sigma \;\leq \mathbb{0}\to\mathbb{1}$ and $(\bigwedge_{j\in J} s\rho_j)\sigma \;\leq \mathrm{dom}((\bigwedge_{i\in I} t\rho_i)\sigma\;)$;
3. Solve the tallying problem for
$$\{(t_1, \mathbb{0}\to\mathbb{1}), (t_1, t_2\to\gamma)\}$$
with $t_1 = \bigwedge_{i\in I} t\rho_i$, $t_2 = \bigwedge_{j\in J} s\rho_j$, and γ fresh
   - if it fails at Step 1, then fail.
   - if it fails at Step 2, then change cardinalities (dove-tail)

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);
2. Split each substitution σ_k (for $k\in I\cup J$) in two: $\sigma_k = \rho_k \circ \sigma'_k$ where ρ_k is a renaming substitution mapping each variable of the domain of σ_k into a fresh variable:
   $(\bigwedge_{i\in I} t\rho_i)\sigma \leq \mathbb{0}\to\mathbb{1}$ and $(\bigwedge_{j\in J} s\rho_j)\sigma \leq \mathrm{dom}((\bigwedge_{i\in I} t\rho_i)\sigma)$;
3. Solve the tallying problem for
   $$\{(t_1, \mathbb{0}\to\mathbb{1}), (t_1, t_2\to\gamma)\}$$
   with $t_1 = \bigwedge_{i\in I} t\rho_i$, $t_2 = \bigwedge_{j\in J} s\rho_j$, and γ fresh
   - if it fails at Step 1, then fail.
   - if it fails at Step 2, then change cardinalities (dove-tail)

➡ **Every solution for γ is a solution for the application.**

Example: map even

Start with the following tallying problem:

$$\{(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq t \to \gamma\}$$

where $t = (\texttt{Int} \to \texttt{Bool}) \wedge (\alpha \setminus \texttt{Int} \to \alpha \setminus \texttt{Int})$ is the type of even

Example: map even

Start with the following tallying problem:

$$\{(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq t \to \gamma\}$$

where $t = (\texttt{Int} \to \texttt{Bool}) \wedge (\alpha \setminus \texttt{Int} \to \alpha \setminus \texttt{Int})$ is the type of `even`

- At step 2 the algorithm generates 9 constraint-sets: one is unsatisfiable ($t \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathbb{0}\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \texttt{Int}\ ,\ \texttt{Bool} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \setminus \texttt{Int}\ ,\ \alpha \setminus \texttt{Int} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \vee \texttt{Int}\ ,\ (\alpha \setminus \texttt{Int}) \vee \texttt{Bool} \leq \beta_1\}$;

Example: map even

Start with the following tallying problem:

$$\{(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq t \to \gamma\}$$

where $t = (\texttt{Int} \to \texttt{Bool}) \wedge (\alpha \setminus \texttt{Int} \to \alpha \setminus \texttt{Int})$ is the type of even

- At step 2 the algorithm generates 9 constraint-sets: one is unsatisfiable ($t \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathbb{0}\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \texttt{Int}\ ,\ \texttt{Bool} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \setminus \texttt{Int}\ ,\ \alpha \setminus \texttt{Int} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \vee \texttt{Int}\ ,\ (\alpha \setminus \texttt{Int}) \vee \texttt{Bool} \leq \beta_1\}$;
- Four solutions for γ:
  $\{\gamma = [\,] \to [\,]\}$,
  $\{\gamma = [\texttt{Int}] \to [\texttt{Bool}]\}$,
  $\{\gamma = [\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]\}$,
  $\{\gamma = [\alpha \vee \texttt{Int}] \to [(\alpha \setminus \texttt{Int}) \vee \texttt{Bool}]\}$.

Example: map even

Start with the following tallying problem:

$$\{(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq t \to \gamma\}$$

where $t = (\mathtt{Int} \to \mathtt{Bool}) \wedge (\alpha \setminus \mathtt{Int} \to \alpha \setminus \mathtt{Int})$ is the type of even

- At step 2 the algorithm generates 9 constraint-sets: one is unsatisfiable ($t \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathbb{0}\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathtt{Int}\ ,\ \mathtt{Bool} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \setminus \mathtt{Int}\ ,\ \alpha \setminus \mathtt{Int} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \vee \mathtt{Int}\ ,\ (\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool} \leq \beta_1\}$;
- Four solutions for γ:
  $\{\gamma = [] \to []\}$,
  $\{\gamma = [\mathtt{Int}] \to [\mathtt{Bool}]\}$,
  $\{\gamma = [\alpha \setminus \mathtt{Int}] \to [\alpha \setminus \mathtt{Int}]\}$,
  $\{\gamma = [\alpha \vee \mathtt{Int}] \to [(\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool}]\}$.
- The last two are minimal and we take their intersection:
  $\{\gamma = ([\alpha \setminus \mathtt{Int}] \to [\alpha \setminus \mathtt{Int}]) \wedge ([\alpha \vee \mathtt{Int}] \to [(\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool}])\}$

Type Reconstruction Algorithm

$$\frac{}{\Gamma \vdash_{\mathcal{R}} c : b_c \rightsquigarrow \{\varnothing\}}\text{(R-CONST)} \qquad \frac{}{\Gamma \vdash_{\mathcal{R}} x : \Gamma(x) \rightsquigarrow \{\varnothing\}}\text{(R-VAR)}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} m_1 : t_1 \rightsquigarrow \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} m_2 : t_2 \rightsquigarrow \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} m_1 m_2 : \alpha \rightsquigarrow \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1 \leq t_2 \to \alpha)\}\}}\text{(R-APPL)}$$

$$\frac{\Gamma, x : \alpha \vdash_{\mathcal{R}} m : t \rightsquigarrow \mathcal{S}}{\Gamma \vdash_{\mathcal{R}} \lambda x.m : \alpha \to \beta \rightsquigarrow \mathcal{S} \sqcap \{\{(t \leq \beta)\}\}}\text{(R-ABSTR)}$$

(R-CASE)

$$\frac{\begin{array}{rl} \mathcal{S} = & (\mathcal{S}_0 \sqcap \{\{(t_0 \leq \mathbb{0})\}\}) \\ \sqcup & (\mathcal{S}_0 \sqcap \mathcal{S}_1 \sqcap \{\{(t_0 \leq t), (t_1 \leq \alpha)\}\}) \\ \sqcup & (\mathcal{S}_0 \sqcap \mathcal{S}_2 \sqcap \{\{(t_0 \leq \neg t), (t_2 \leq \alpha)\}\}) \\ \sqcup & (\mathcal{S}_0 \sqcap \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1 \vee t_2 \leq \alpha)\}\}) \end{array} \\ \Gamma \vdash_{\mathcal{R}} m_0 : t_0 \rightsquigarrow \mathcal{S}_0 \qquad \Gamma \vdash_{\mathcal{R}} m_1 : t_1 \rightsquigarrow \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} m_2 : t_2 \rightsquigarrow \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} (m_0 \in t \,?\, m_1 : m_2) : \alpha \rightsquigarrow \mathcal{S}}$$

where α, α_i and β in each rule are fresh type variables.

Semantic subtyping with type variables

The subtyping relation is decidable in EXPTIME.

Semantic subtyping with type variables

The subtyping relation is decidable in EXPTIME.

We can prove relevant relations on infinite types, *eg*., for the type of generic α-lists:

$$[\alpha] \stackrel{\text{def}}{=} \mu z.(\alpha \times z) \vee \text{nil}$$

Semantic subtyping with type variables

The subtyping relation is decidable in EXPTIME.

We can prove relevant relations on infinite types, *eg*., for the type of generic α-lists:

$$[\alpha] \stackrel{\text{def}}{=} \mu z.(\alpha \times z) \vee \text{nil}$$

we can prove that it contains both the α-lists of even length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee \text{nil}}_{\alpha\text{-lists of even length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

and the α-lists with of odd length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \text{nil})}_{\alpha\text{-lists of odd length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

Semantic subtyping with type variables

The subtyping relation is decidable in EXPTIME.

We can prove relevant relations on infinite types, *eg*., for the type of generic α-lists:

$$[\alpha] \stackrel{\text{def}}{=} \mu z.(\alpha \times z) \vee \text{nil}$$

we can prove that it contains both the α-lists of even length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee \text{nil}}_{\alpha\text{-lists of even length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

and the α-lists with of odd length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \text{nil})}_{\alpha\text{-lists of odd length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

and that it is itself contained in the union of the two, that is:

$$[\alpha] \sim (\mu z.(\alpha \times (\alpha \times z)) \vee \text{nil}) \vee (\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \text{nil}))$$

Axiomatic properties of intersection types are here *deduced* from the semantic interpretation:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \sim \alpha \vee \beta \to \gamma$$

Axiomatic properties of intersection types are here *deduced* from the semantic interpretation:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \sim \alpha \vee \beta \to \gamma$$

as well as classic distributivity laws:

$$(\alpha \vee \beta \times \gamma) \sim (\alpha \times \gamma) \vee (\beta \times \gamma)$$

Axiomatic properties of intersection types are here *deduced* from the semantic interpretation:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \sim \alpha \vee \beta \to \gamma$$

as well as classic distributivity laws:

$$(\alpha \vee \beta \times \gamma) \sim (\alpha \times \gamma) \vee (\beta \times \gamma)$$

Most importantly we can use standard set-theoretic laws to show:

- that every type is equivalent to a type in disjunctive normal form
- to *deduce* decomposition rules used in algorithms such as

$$s_1 \times s_2 \leq t_1 \times t_2 \iff \big(s_1 \leq \mathbb{0} \text{ or } s_2 \leq \mathbb{0} \text{ or } (s_1 \leq t_1 \text{ and } s_2 \leq t_2) \big)$$

Expressions

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e \mid (e, e) \mid \pi_i e$$

Expressions

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e \mid (e, e) \mid \pi_i e$$

Why a type-case:

Why explicitly-typed functions:

Expressions

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e \mid (e,e) \mid \pi_i e$$

Why a type-case:

- Intersection types with "real" overloading vs. coherent one
  [*eg*, non diverging functions in $(\texttt{Int}{\to}\texttt{Bool}) \wedge (\texttt{Bool}{\to}\texttt{Int})$]

Why explicitly-typed functions:

Expressions

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e \mid (e,e) \mid \pi_i e$$

Why a type-case:

- Intersection types with "real" overloading vs. coherent one
  [*eg*, non diverging functions in $(\texttt{Int}\to\texttt{Bool}) \wedge (\texttt{Bool}\to\texttt{Int})$]
- The following containment is *strict*:

$$s_1 \vee s_2 \to t_1 \wedge t_2 \quad \lneq \quad (s_1 \to t_1) \wedge (s_2 \to t_2)$$

Why explicitly-typed functions:

Expressions

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e\in t\,?\,e : e \mid (e, e) \mid \pi_i e$$

Why a type-case:

- Intersection types with "real" overloading vs. coherent one
  [*eg*, non diverging functions in $(\texttt{Int}\to\texttt{Bool}) \wedge (\texttt{Bool}\to\texttt{Int})$]
- The following containment is *strict*:

$$s_1 \vee s_2 \to t_1 \wedge t_2 \quad \lneq \quad (s_1 \to t_1) \wedge (s_2 \to t_2)$$

Why explicitly-typed functions: [a consequence of the type-case]

Avoid paradoxes:

$$\mu f.\lambda x.f\in(\mathbb{1}\to\texttt{Int})\,?\,\texttt{true} : 42$$

It has type $\mathbb{1}\to\texttt{Int}$ iff it *does not* have type $\mathbb{1}\to\texttt{Int}$.

Expressions

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t \,?\, e : e \mid (e,e) \mid \pi_i e$$

Why a type-case:

- Intersection types with "real" overloading vs. coherent one
  [*eg*, non diverging functions in $(\texttt{Int}{\to}\texttt{Bool}) \wedge (\texttt{Bool}{\to}\texttt{Int})$]
- The following containment is *strict*:

$$s_1 \vee s_2 \to t_1 \wedge t_2 \quad \lneq \quad (s_1 \to t_1) \wedge (s_2 \to t_2)$$

Why explicitly-typed functions: [a consequence of the type-case]

Avoid paradoxes:

$$\mu f.\lambda x.f{\in}(\mathbb{1}{\to}\texttt{Int})\,?\,\texttt{true} : 42$$

It has type $\mathbb{1}{\to}\texttt{Int}$ iff it *does not* have type $\mathbb{1}{\to}\texttt{Int}$.

- Explicitly assign the type $\mathbb{1} \to \texttt{Int}{\vee}\texttt{Bool}$ to it.
- More expressive with the result type (type of x not enough)

How to type-annotate functions?

$$\lambda x.(x{\in}\texttt{Int}\,?\,\texttt{true} : 42)$$

How to type-annotate functions?

$$\lambda x.(x \in \texttt{Int} \, ? \, \texttt{true} : 42)$$

It has type $(\texttt{Int} \to \texttt{Bool}) \wedge (\neg\texttt{Int} \to \texttt{Int})$ but we will be content with $(\texttt{Int} \to \texttt{Bool}) \wedge (\texttt{Bool} \to \texttt{Int})$

How to type-annotate functions?

$$\lambda x^{t?}.(x\in\mathtt{Int}\ ?\ \mathtt{true} : 42)$$

It has type $(\mathtt{Int}\to\mathtt{Bool})\wedge(\neg\mathtt{Int}\to\mathtt{Int})$ but we will be content with $(\mathtt{Int}\to\mathtt{Bool})\wedge(\mathtt{Bool}\to\mathtt{Int})$

- Church style?

How to type-annotate functions?

$$\lambda x^{\texttt{Int}\vee\texttt{Bool}}.(x\in\texttt{Int}\,?\,\texttt{true} : 42)$$

It has type $(\texttt{Int}\to\texttt{Bool})\wedge(\neg\texttt{Int}\to\texttt{Int})$ but we will be content with

$$(\texttt{Int}\to\texttt{Bool})\wedge(\texttt{Bool}\to\texttt{Int})$$

- Church style?

  If we assign $\texttt{Int}\vee\texttt{Bool}$ to x the type , we can only deduce

$$\texttt{Int}\vee\texttt{Bool}\to\texttt{Int}\vee\texttt{Bool}$$

How to type-annotate functions?

$$\lambda x.(x{\in}\mathtt{Int}\,?\,\mathtt{true} : 42)$$

It has type $(\mathtt{Int}{\to}\mathtt{Bool}){\wedge}(\neg\mathtt{Int}{\to}\mathtt{Int})$ but we will be content with

$$(\mathtt{Int} \to \mathtt{Bool}){\wedge}(\mathtt{Bool} \to \mathtt{Int})$$

- Church style?
  If we assign $\mathtt{Int}{\vee}\mathtt{Bool}$ to x the type , we can only deduce
  $$\mathtt{Int}{\vee}\mathtt{Bool} \to \mathtt{Int}{\vee}\mathtt{Bool}$$
- $\mathbb{C}$Duce solution: annotate λ's with their intersection type

How to type-annotate functions?

$$\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\texttt{Bool}\to\texttt{Int})}x.\,(x{\in}\texttt{Int}\,?\,\texttt{true} : 42)$$

It has type $(\texttt{Int}\to\texttt{Bool})\wedge(\neg\texttt{Int}\to\texttt{Int})$ but we will be content with

$$(\texttt{Int} \to \texttt{Bool})\wedge(\texttt{Bool} \to \texttt{Int})$$

- Church style?
  If we assign $\texttt{Int}\vee\texttt{Bool}$ to x the type , we can only deduce
  $$\texttt{Int}\vee\texttt{Bool} \to \texttt{Int}\vee\texttt{Bool}$$
- $\mathbb{C}$Duce solution: annotate λ's with their intersection type

How to type-annotate functions?

$$\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\texttt{Bool}\to\texttt{Int})}x.\,(x\in\texttt{Int}\,?\,\texttt{true} : 42)$$

It has type $(\texttt{Int}\to\texttt{Bool})\wedge(\neg\texttt{Int}\to\texttt{Int})$ but we will be content with

$$(\texttt{Int} \to \texttt{Bool})\wedge(\texttt{Bool} \to \texttt{Int})$$

- Church style?
  If we assign $\texttt{Int}\vee\texttt{Bool}$ to x the type , we can only deduce
  $$\texttt{Int}\vee\texttt{Bool} \to \texttt{Int}\vee\texttt{Bool}$$
- $\mathbb{C}$Duce solution: annotate λ's with their intersection type

Syntax for λ-abstractions

Add to expressions

$$\lambda^{\wedge_{i\in I} s_i\to t_i}x.e$$

Well typed if from $x : s_i$ we can deduce $e : t_i$, for all $i\in I$.