

# Typing the Y Combinator

---

Esther Wang

LambdaConf 2018

# Table of contents

1. Why have recursive types?
2. Examples of recursive types
3. Fixpoints
4. Typing the Y combinator
5. Equi vs. iso-recursive types
6. Recursive types in Haskell
7. Some additional theory

# Why have recursive types?

---

# The Y combinator

$$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$$

# The Y combinator

$$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$$

What if we want static types?

## Omega: a smaller example

$\lambda x . x x$

## Omega: a smaller example

$$\lambda x . x x$$

1.  $x$  must be a function

## Omega: a smaller example

$$\lambda x . x x$$

1.  $x$  must be a function
2.  $x :: a \rightarrow b$



## Omega: a smaller example

$$\lambda x . x x$$

1.  $x$  must be a function
2.  $x :: a \rightarrow b$
3.  $a$  must be the type of  $x$ , so  $a = a \rightarrow b$

## Omega: a smaller example

$$\lambda x . x x$$

1.  $x$  must be a function
2.  $x :: a \rightarrow b$
3.  $a$  must be the type of  $x$ , so  $a = a \rightarrow b$
4.  $a \rightarrow b = (a \rightarrow b) \rightarrow b = ((a \rightarrow b) \rightarrow b) \rightarrow b = \dots$

# Simple types are insufficient

Some terms from the untyped lambda calculus cannot be expressed in the simply-typed lambda calculus

$$\lambda x. x x$$

# Simple types are insufficient

Some terms from the untyped lambda calculus cannot be expressed in the simply-typed lambda calculus

$$\lambda x . x x$$
$$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$$

# Simple types are insufficient

Some terms from the untyped lambda calculus cannot be expressed in the simply-typed lambda calculus

$$\lambda x . x x$$
$$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$$

Solution: recursive types

## Examples of recursive types

---

## Recursive types are ubiquitous

```
data Nat = Zero | Succ Nat
data IntList = Nil | Cons Int IntList
data StringTree = Leaf String | Node StringTree StringTree
```

- Recursive types have the form  $\mu a. F\ a$ , where  $F$  is a type expression



# Recursive types in type theory

- Recursive types have the form  $\mu a. F\ a$ , where  $F$  is a type expression
- $\mu$  is the type-level **fixpoint operator**

# Recursive types in type theory

- Recursive types have the form  $\mu a. F\ a$ , where  $F$  is a type expression
- $\mu$  is the type-level **fixpoint operator**
- Adding  $\mu$  to our type system allows us to type any term from the untyped lambda calculus

## Example: Nat

```
data Nat = Zero | Succ Nat
```

## Example: Nat

```
data Nat = Zero | Succ Nat
```

1. `Nat` should satisfy the type equation  $\text{Nat} = 1 + \text{Nat}$

## Example: Nat

```
data Nat = Zero | Succ Nat
```

1. `Nat` should satisfy the type equation  $\text{Nat} = 1 + \text{Nat}$
2.  $\text{Nat} = \mu a . 1 + a$

## Example: IntList

```
data IntList = Nil | Cons Int IntList
```

## Example: IntList

```
data IntList = Nil | Cons Int IntList
```

```
IntList =  $\mu$  a . 1 + Int * a
```

## Example: Variadic Functions

A *variadic function* accepts any number of arguments. For example:

```
sumAllInts 1
```

```
--> 1
```

```
sumAllInts 1 2 3
```

```
--> 6
```



## Example: Variadic Functions

A *variadic function* accepts any number of arguments. For example:

```
sumAllInts 1
```

```
--> 1
```

```
sumAllInts 1 2 3
```

```
--> 6
```

```
1. sumAllInts :: Int -> ???
```

## Example: Variadic Functions

A *variadic function* accepts any number of arguments. For example:

```
sumAllInts 1
--> 1
sumAllInts 1 2 3
--> 6
```

1. `sumAllInts :: Int -> ???`
2. `sumAllInts :: Int -> (Int + ???)`

## Example: Variadic Functions

A *variadic function* accepts any number of arguments. For example:

```
sumAllInts 1
--> 1
sumAllInts 1 2 3
--> 6
```

1. `sumAllInts :: Int -> ???`
2. `sumAllInts :: Int -> (Int + ???)`
3. `sumAllInts ::  $\mu$  a . Int -> (Int + a)`

# Fixpoints

---

# Definitions

Fixed point  $x$

$$x = f\ x = f\ (f\ x) = \dots$$

Fixpoint combinator **fix**

$$\text{fix } f = x$$

Combined definitions

$$\text{fix } f = f\ (\text{fix } f)$$

# Fixpoint combinators

- The Y Combinator is **fix** on the **term** level
- $\mu$  is **fix** on the **type** level

- Given a recursive type

$\mu a. F\ a$

- Given a recursive type

$$\mu a. F a$$

- By the definition of a fixpoint combinator,  $\mu F = x$ , where  $x$  is a fixed point



- Given a recursive type

$$\mu a. F a$$

- By the definition of a fixpoint combinator,  $\mu F = x$ , where  $x$  is a fixed point
- By the definition of fixed points,  $x = F x$

# Defining $\mu$

- Given a recursive type

$$\mu a. F a$$

- By the definition of a fixpoint combinator,  $\mu F = x$ , where  $x$  is a fixed point
- By the definition of fixed points,  $x = F x$
- Substituting for  $x$ ,

$$\mu F = F (\mu F)$$

# Defining $\mu$

$\mu$  is defined as

$$\mu F = F (\mu F)$$

You can substitute the recursive type into itself

$$\text{Nat} = \mu a . 1 + a = 1 + (\mu a . 1 + a) = \dots$$

## Typing the Y combinator

---

# Typing the Omega combinator

$$\lambda x . x x$$

1.  $x :: a \rightarrow b$
2.  $a$  must be the type of  $x$ , so  $a = a \rightarrow b$

# Typing the Omega combinator

$$\lambda x . x x$$

1.  $x :: a \rightarrow b$
2.  $a$  must be the type of  $x$ , so  $a = a \rightarrow b$
3.  $a = \mu a . a \rightarrow b$

# Typing the Omega combinator

$$\lambda x . x x$$

1.  $x :: a \rightarrow b$
2.  $a$  must be the type of  $x$ , so  $a = a \rightarrow b$
3.  $a = \mu a . a \rightarrow b$
4.  $x :: \mu a . a \rightarrow b$

$$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$$



## Typing the Y combinator

$$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$$

1.  $f :: a \rightarrow b$

$$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$$

1.  $f :: a \rightarrow b$
2.  $x :: c \rightarrow a$

# Typing the Y combinator

$$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$$

1.  $f :: a \rightarrow b$
2.  $x :: c \rightarrow a$
3.  $c = c \rightarrow a$ , so  $c = \mu c . c \rightarrow a$

# Typing the Y combinator

$$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$$

1.  $f :: a \rightarrow b$
2.  $x :: c \rightarrow a$
3.  $c = c \rightarrow a$ , so  $c = \mu c . c \rightarrow a$
4. The Y combinator has type  $(a \rightarrow b) \rightarrow b$ , for fixed  $a$  and  $b$

# Typing the Y combinator

$$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$$

1.  $f :: a \rightarrow b$
2.  $x :: c \rightarrow a$
3.  $c = c \rightarrow a$ , so  $c = \mu c . c \rightarrow a$
4. The Y combinator has type  $(a \rightarrow b) \rightarrow b$ , for fixed  $a$  and  $b$
5. There is one more constraint, which is that  $a = b$ . Can you figure out why?

## Equi vs. iso-recursive types

---

- In our examples so far, we've implicitly used

$$\mu F = F (\mu F)$$

# Equi-recursive types

- In our examples so far, we've implicitly used

$$\mu F = F (\mu F)$$

- **equi-recursive** approach, where a recursive type is interchangeable with its expansion



# Equi-recursive types

- In our examples so far, we've implicitly used

$$\mu F = F (\mu F)$$

- **equi-recursive** approach, where a recursive type is interchangeable with its expansion
- Easy to add to a type system

# Equi-recursive types

- In our examples so far, we've implicitly used

$$\mu F = F (\mu F)$$

- **equi-recursive** approach, where a recursive type is interchangeable with its expansion
- Easy to add to a type system
- Difficult to implement in a typechecker

- In the **iso-recursive** approach,

$$\mu F \sim F (\mu F)$$

- In the **iso-recursive** approach,

$$\mu F \sim F (\mu F)$$

- A recursive type and its expansion are *isomorphic*

- In the **iso-recursive** approach,

$$\mu F \sim F (\mu F)$$

- A recursive type and its expansion are *isomorphic*
- The functions `roll` and `unroll` witness the isomorphism

Let  $S = \mu a . T$ , where  $T = F a$

$$\text{unroll}[S] :: S \rightarrow [a \mapsto S] T$$

Let  $S = \mu a . T$ , where  $T = F a$

$$\begin{aligned}\text{unroll}[S] &:: S \rightarrow [a \mapsto S] T \\ \text{roll}[S] &:: [a \mapsto S] T \rightarrow S\end{aligned}$$

Roll and unroll are inverses

$$\text{unroll}[S] (\text{roll}[T] (e)) \rightarrow e$$



- Haskell and OCaml use iso-recursive types

# Use in production languages

- Haskell and OCaml use iso-recursive types
- `roll` and `unroll` are baked into constructors and pattern matching

# Use in production languages

- Haskell and OCaml use iso-recursive types
- `roll` and `unroll` are baked into constructors and pattern matching
- In Java, class definitions implicitly `roll`, and calling a method uses `unroll`

# Use in production languages

- Haskell and OCaml use iso-recursive types
- `roll` and `unroll` are baked into constructors and pattern matching
- In Java, class definitions implicitly `roll`, and calling a method uses `unroll`
- The functions `roll` and `unroll` witness the isomorphism

# Recursive types in Haskell

---

Haskell has recursion, so no need to use the Y combinator

```
fix f = f (fix f)
```

Haskell has recursion, so no need to use the Y combinator

```
fix f = f (fix f)
```

If this feature didn't exist, would Haskell still be Turing-complete?

# Naive implementation of the Y combinator

```
Prelude> y = \f -> (\x -> f (x x)) (\x -> f (x x))
```

```
<interactive>:7:23: error:
```

- Occurs check: cannot construct the infinite type:

$t_0 \sim t_0 \rightarrow t$

Expected type:  $t_0 \rightarrow t$

Actual type:  $(t_0 \rightarrow t) \rightarrow t$

- In the first argument of 'x', namely 'x'

In the first argument of 'f', namely '(x x)'

In the expression:  $f (x x)$

- Relevant bindings include

$x :: (t_0 \rightarrow t) \rightarrow t$  (bound at <interactive>:7:13)

$f :: t \rightarrow t$  (bound at <interactive>:7:6)

$y :: (t \rightarrow t) \rightarrow t$  (bound at <interactive>:7:1)



# Implementing the Y combinator

Define  $\mu F$

```
newtype Mu f = Mu (f (Mu f))
unroll (Mu f) = f
roll = Mu
```

## Remember the types

$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$

1.  $f :: a \rightarrow b$
2.  $x :: \mu c . c \rightarrow a$

## Remember the types

$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$

1.  $f :: a \rightarrow b$
2.  $x :: \mu c . c \rightarrow a$
3. We need to be able to write the type of  $x$  in terms of  $\mu a.F a$

## Remember the types

$\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$

1.  $f :: a \rightarrow b$
2.  $x :: \mu c . c \rightarrow a$
3. We need to be able to write the type of  $x$  in terms of  $\mu a.F a$
4.  $x :: \mu c . F(c)$ , where  $F(c) = c \rightarrow a$  for some fixed  $a$

# Implementing the Y combinator

Define  $F(c) = c \rightarrow a$

```
newtype Mu f = Mu (f (Mu f))
unroll (Mu f) = f
roll = Mu
```

```
newtype F' c a = F' (c -> a)
unF (F' f) = f
type F c = Mu (F' c)
```

# Implementing the Y combinator

Define  $F(c) = c \rightarrow a$

```
newtype Mu f = Mu (f (Mu f))
unroll (Mu f) = f
roll = Mu
```

```
newtype F' c a = F' (a -> c)
unF (F' f) = f
type F c = Mu (F' c)
unroll' = unF . unroll
roll'   = roll . F'
```

# Implementing the Y combinator

```
newtype Mu f = Mu (f (Mu f))
unroll (Mu f) = f
roll = Mu
```

```
newtype F' c a = F' (a -> c)
unF (F' f) = f
type F c = Mu (F' c)
unroll' = unF . unroll
roll'   = roll . F'
```

```
y f = (\x -> f (unroll' x x))
      $ roll' (\x -> f (unroll' x x))
```

`Mu` is the same as `Fix` from the *recursion-schemes* library!

```
newtype Fix f = Fix (f (Fix f))
```

```
unfix (Fix f) = f
```

```
fix = Fix
```



## Some additional theory

---

- Data consists of indefinitely large, but finite structures

- Data consists of indefinitely large, but finite structures
  - For example, finite lists

- Data consists of indefinitely large, but finite structures
  - For example, finite lists
  - Data is defined by constructors

- Data consists of indefinitely large, but finite structures
  - For example, finite lists
  - Data is defined by constructors
  - **Cons** allows us to build a bigger list using an element and a given list

- Data consists of indefinitely large, but finite structures
  - For example, finite lists
  - Data is defined by constructors
  - **Cons** allows us to build a bigger list using an element and a given list
  - Use structural induction for proofs

# Data and codata

- Data consists of indefinitely large, but finite structures
  - For example, finite lists
  - Data is defined by constructors
  - **Cons** allows us to build a bigger list using an element and a given list
  - Use structural induction for proofs
- Codata consists of data, but also includes potentially infinite structures

# Data and codata

- Data consists of indefinitely large, but finite structures
  - For example, finite lists
  - Data is defined by constructors
  - **Cons** allows us to build a bigger list using an element and a given list
  - Use structural induction for proofs
- Codata consists of data, but also includes potentially infinite structures
  - For example, streams



# Data and codata

- Data consists of indefinitely large, but finite structures
  - For example, finite lists
  - Data is defined by constructors
  - **Cons** allows us to build a bigger list using an element and a given list
  - Use structural induction for proofs
- Codata consists of data, but also includes potentially infinite structures
  - For example, streams
  - Codata is defined by destructors

# Data and codata

- Data consists of indefinitely large, but finite structures
  - For example, finite lists
  - Data is defined by constructors
  - **Cons** allows us to build a bigger list using an element and a given list
  - Use structural induction for proofs
- Codata consists of data, but also includes potentially infinite structures
  - For example, streams
  - Codata is defined by destructors
  - **Head** and **Tail** allow us to get an element and a new stream, given a stream

# Data and codata

- Data consists of indefinitely large, but finite structures
  - For example, finite lists
  - Data is defined by constructors
  - **Cons** allows us to build a bigger list using an element and a given list
  - Use structural induction for proofs
- Codata consists of data, but also includes potentially infinite structures
  - For example, streams
  - Codata is defined by destructors
  - **Head** and **Tail** allow us to get an element and a new stream, given a stream
  - Use coinduction for proofs

- Recall that  $\text{IntList} = \mu a . 1 + \text{Int} * a$

---

<sup>1</sup>See the Knaster-Tarski Theorem for a rigorous definition

# Least fixed points

- Recall that  $\text{IntList} = \mu a . 1 + \text{Int} * a$
- $F a = 1 + \text{Int} * a$

---

<sup>1</sup>See the Knaster-Tarski Theorem for a rigorous definition

# Least fixed points

- Recall that  $\text{IntList} = \mu a . 1 + \text{Int} * a$
- $F a = 1 + \text{Int} * a$
- The type of finite integer lists is the **least fixed point** of  $F$

---

<sup>1</sup>See the Knaster-Tarski Theorem for a rigorous definition

# Least fixed points

- Recall that  $\text{IntList} = \mu a . 1 + \text{Int} * a$
- $F a = 1 + \text{Int} * a$
- The type of finite integer lists is the **least fixed point** of  $F$
- The least fixed point is the least set  $X$  for which  $X = F X$ <sup>1</sup>

---

<sup>1</sup>See the Knaster-Tarski Theorem for a rigorous definition

# Least fixed points

- Recall that  $\text{IntList} = \mu a . 1 + \text{Int} * a$
- $F a = 1 + \text{Int} * a$
- The type of finite integer lists is the **least fixed point** of  $F$
- The least fixed point is the least set  $X$  for which  $X = F X$ <sup>1</sup>
- All elements of  $X$  can be generated by  $F$

---

<sup>1</sup>See the Knaster-Tarski Theorem for a rigorous definition



# Least fixed points

- Recall that  $\text{IntList} = \mu a . 1 + \text{Int} * a$
- $F a = 1 + \text{Int} * a$
- The type of finite integer lists is the **least fixed point** of  $F$
- The least fixed point is the least set  $X$  for which  $X = F X$ <sup>1</sup>
- All elements of  $X$  can be generated by  $F$

---

<sup>1</sup>See the Knaster-Tarski Theorem for a rigorous definition

# Greatest fixed points

•  $\text{IntStream} = \mu a . \text{Int} * a$

---

<sup>2</sup>See the Knaster-Tarski Theorem for a rigorous definition

# Greatest fixed points

- $\text{IntStream} = \mu a . \text{Int} * a$
- $F a = \text{Int} * a$

---

<sup>2</sup>See the Knaster-Tarski Theorem for a rigorous definition

# Greatest fixed points

- $\text{IntStream} = \mu a . \text{Int} * a$
- $F a = \text{Int} * a$
- The type of integer streams is the **greatest fixed point** of  $F$

---

<sup>2</sup>See the Knaster-Tarski Theorem for a rigorous definition

# Greatest fixed points

- `IntStream =  $\mu$  a . Int * a`
- `F a = Int * a`
- The type of integer streams is the **greatest fixed point** of `F`
- The greatest fixed point is the greatest set  $X$  for which  $X = F X^2$

---

<sup>2</sup>See the Knaster-Tarski Theorem for a rigorous definition

- The typechecking algorithm for equi-recursive types works by determining whether a type is a member of a recursive type's least/greatest fixed point

- The typechecking algorithm for equi-recursive types works by determining whether a type is a member of a recursive type's least/greatest fixed point
- For category theorists, data is an initial  $F$ -algebra, codata is a terminal  $F$ -coalgebra

## Conclusion

---



- The type operator  $\mu$  allows us to type recursive data and terms

# Conclusion

- The type operator  $\mu$  allows us to type recursive data and terms
- $\mu$  is defined as a type-level fixpoint combinator:  $\mu a.F a$

# Conclusion

- The type operator  $\mu$  allows us to type recursive data and terms
- $\mu$  is defined as a type-level fixpoint combinator:  $\mu a.F a$
- The equi-recursive approach treats a recursive type as *equal* to its expansion

# Conclusion

- The type operator  $\mu$  allows us to type recursive data and terms
- $\mu$  is defined as a type-level fixpoint combinator:  $\mu a.F a$
- The equi-recursive approach treats a recursive type as *equal* to its expansion
- The iso-recursive approach treats a recursive type as *isomorphic* to its expansion

## Further reading

1. *Types and Programming Languages* by Benjamin Pierce
2. “Recursive types for free!” by Philip Wadler
3. Recursion schemes
4. Fixpoints and iso-recursive types
5. Data and codata

## Questions?

