Recursive Types

Advanced Compiler Construction and Program Analysis

Lecture 10

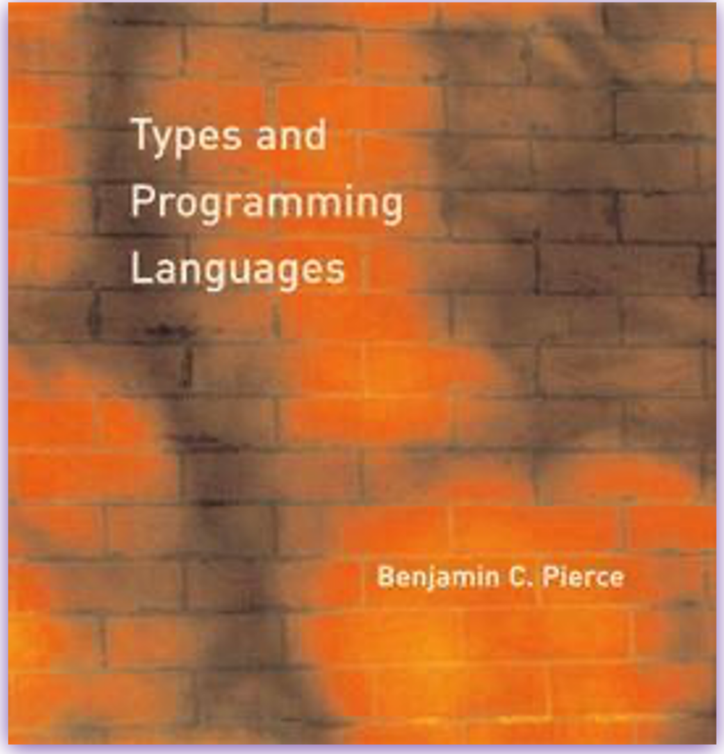
Innopolis University, Spring 2022

The topics of this lecture are covered in detail in...

Benjamin C. Pierce.

Types and Programming Languages

MIT Press 2002



A list of natural numbers

Previously, we have implemented lists using built-in support for `List[T]`. Ignoring the generic part, can we define a list of numbers as a type alias?

```
NatList = <nil: Unit, cons: {…, …}>
```

A list of natural numbers

Previously, we have implemented lists using built-in support for `List[T]`. Ignoring the generic part, can we define a list of numbers as a type alias?

```
NatList = <nil: Unit, cons: {Nat, …}>
```

A list of natural numbers

Previously, we have implemented lists using built-in support for `List[T]`. Ignoring the generic part, can we define a list of numbers as a type alias?

```
NatList = <nil: Unit, cons: {Nat,
NatList}>
```

A list of natural numbers: unrolling definition

```
NatList = <nil: Unit, cons: {Nat,
NatList}>
```

A list of natural numbers: unrolling definition

```
NatList = <nil: Unit, cons: {Nat,
NatList}>
    <nil: •, cons: •>
```

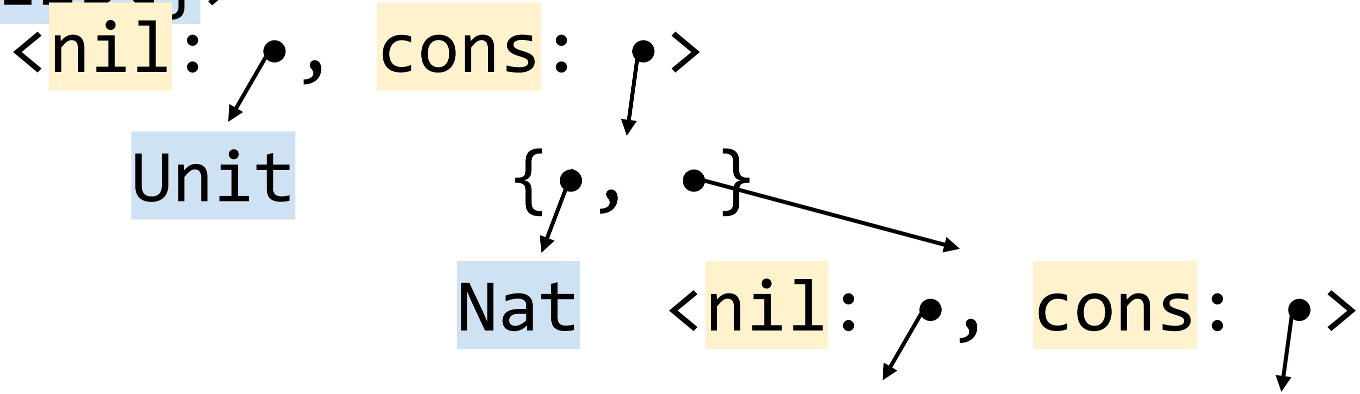
A list of natural numbers: unrolling definition

```
NatList = <nil: Unit, cons: {Nat,
NatList}>
    <nil: •, cons: •>

        Unit        {•, •}
```

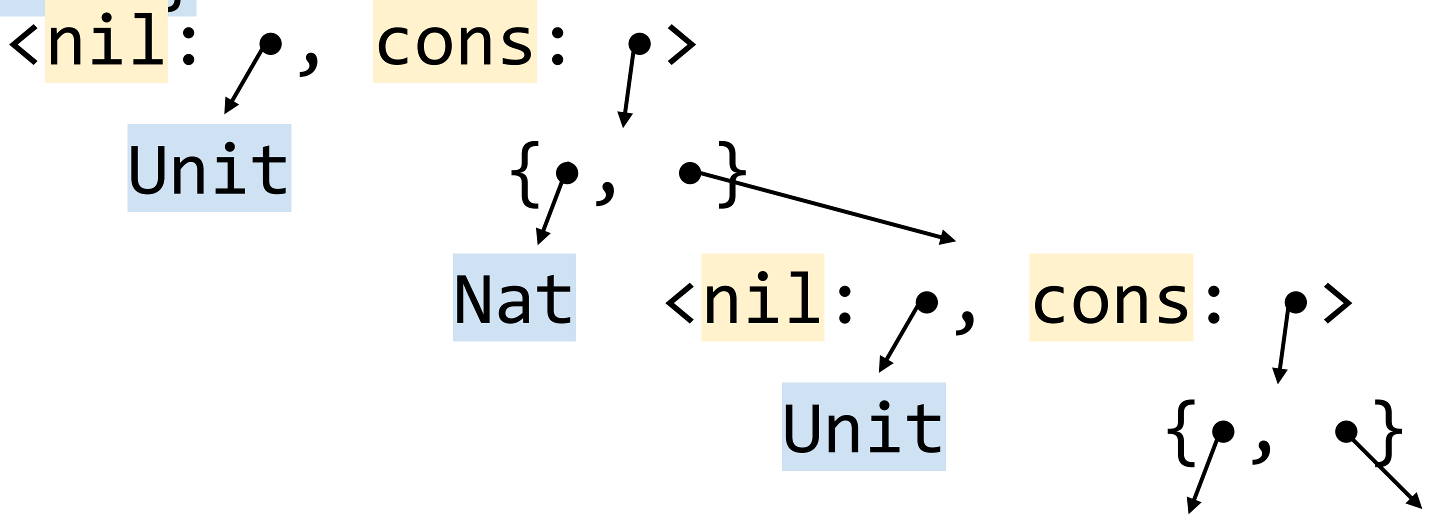
A list of natural numbers: unrolling definition

```
NatList = <nil: Unit, cons: {Nat, NatList}>
```



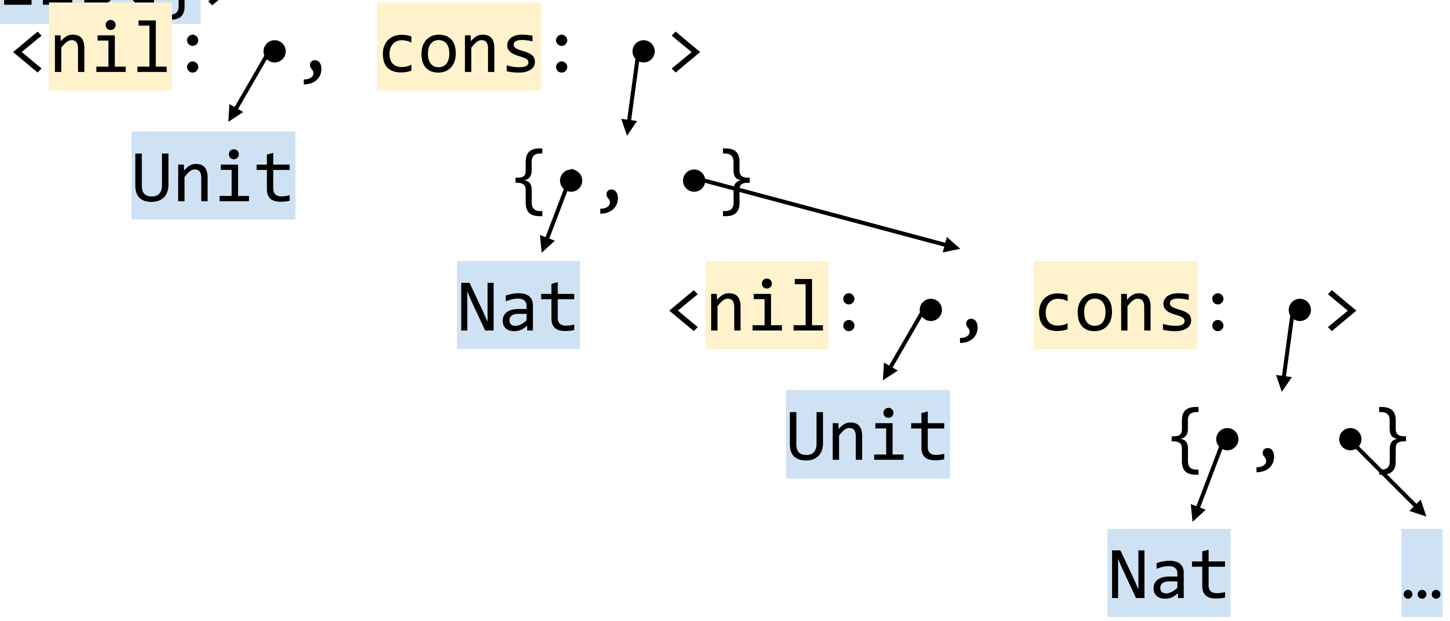
A list of natural numbers: unrolling definition

```
NatList = <nil: Unit, cons: {Nat, NatList}>
```



A list of natural numbers: unrolling definition

```
NatList = <nil: Unit, cons: {Nat, NatList}>
```



A list of natural numbers: fixpoint combinator

```
letrec factorial : Nat → Nat
      = λn. if n == 0 then 1 else n * factorial (n-
1)
```

A list of natural numbers: fixpoint combinator

```
letrec factorial : Nat → Nat
      = λn. if n == 0 then 1 else n * factorial (n-
1)
```

```
let factorial : Nat → Nat
      = fix λf.
            λn. if n == 0 then 1 else n * f (n-1)
```

A list of natural numbers: fixpoint combinator

```
letrec factorial : Nat → Nat
      = λn. if n == 0 then 1 else n * factorial (n-
1)
```

```
let factorial : Nat → Nat
      = fix λf.
            λn. if n == 0 then 1 else n * f (n-1)
```

```
NatList = <nil: Unit, cons: {Nat, NatList}>
```

A list of natural numbers: fixpoint combinator

```
letrec factorial : Nat → Nat
      = λn. if n == 0 then 1 else n * factorial (n-
1)

let factorial : Nat → Nat
      = fix λf.
            λn. if n == 0 then 1 else n * f (n-1)


NatList = <nil: Unit, cons: {Nat, NatList}>

NatList = μX. <nil: Unit, cons: {Nat, X}>
```

A list of natural numbers: examples

```
NatList = μX. <nil: Unit, cons: {Nat, X}>
```

A list of natural numbers: examples

```
NatList = μX. <nil: Unit, cons: {Nat, X}>

let nil : NatList =
      <nil = unit> as NatList
```

A list of natural numbers: examples

```
NatList = μX. <nil: Unit, cons: {Nat, X}>

let nil : NatList =
      <nil = unit> as NatList

let cons : Nat → NatList → NatList =
      λn:Nat.λl:NatList. <cons = {n, l}> as
NatList
```

A list of natural numbers: examples

```
NatList = μX. <nil: Unit, cons: {Nat, X}>

let nil : NatList =
      <nil = unit> as NatList

let cons : Nat → NatList → NatList =
      λn:Nat.λl:NatList. <cons = {n, l}> as
NatList

let isnil : NatList → Bool =
      λl:NatList. case l of
         <nil  = _>          ⇒ true
       | <cons = {_, _}>     ⇒ false
```

A list of natural numbers: exercise

```
NatList = <nil: Unit, cons: {Nat, NatList}>
NatList = μX. <nil: Unit, cons: {Nat, X}>
```

Exercise 10.1. Assuming `plus : Nat → Nat → Nat`, implement recursive function

```
sumList : NatList → Nat
```

Hungry functions

```
Hungry = μX. Nat → X
```

Hungry functions

```
Hungry = μX. Nat → X

let g : Hungry =
      fix (λf:Nat→Hungry.λx:Nat.f)
in g 0 1 2 3 4 5
```

Streams

```
Stream = μX. Unit → {Nat, X}
```

Streams

```
Stream = μX. Unit → {Nat, X}

let head : Stream → Nat =
      λs:Stream.(s unit).1
```

Streams

```
Stream = μX. Unit → {Nat, X}

let head : Stream → Nat =
      λs:Stream.(s unit).1

let tail : Stream → Stream =
      λs:Stream.(s unit).2
```

Streams

```
Stream = μX. Unit → {Nat, X}

let head : Stream → Nat =
      λs:Stream.(s unit).1

let tail : Stream → Stream =
      λs:Stream.(s unit).2

letrec upfrom : Nat→Stream =
      λn:Nat. λ_:Unit. {n, upfrom (succ n)}
```

Streams: exercise

`Stream = μX. Unit → {Nat, X}`

Exercise 10.2. Assuming `plus : Nat → Nat → Nat`, define a stream of Fibonacci numbers (1, 1, 2, 3, 5, 8, ...):

`fib : Stream`

Processes

```
Process = μX. Nat → {Nat, X}
```

Processes

```
Process = μX. Nat → {Nat, X}

letrec sumProcessFrom : Nat → Process =
      λacc:Nat.λn:Nat.
            let newacc = plus acc n
             in {newacc, sumProcessFrom
newacc}
```

Processes

```
Process = μX. Nat → {Nat, X}

letrec sumProcessFrom : Nat → Process =
    λacc:Nat.λn:Nat.
        let newacc = plus acc n
         in {newacc, sumProcessFrom
newacc}

let sumProcess : Process
    = sumProcessFrom 0
```

Purely Functional Objects

```
Counter = μX. {get: Nat, inc: Unit → X}
```

Purely Functional Objects

```
Counter = μX. {get: Nat, inc: Unit → X}

letrec newCounter : {x: Nat} → Counter =
      λrep:{x: Nat}.
            { get = rep.x
            , inc = λ_:Unit.
                        newCounter {x = succ
(rep.x)}
            }
```

Well-typed fixed point combinator

Untyped fixed point:

```
fix = λf.(λx.f (x x))(λx.f (x x))
```

Well-typed fixed point combinator

Untyped fixed point:

```
fix = λf.(λx.f (x x))(λx.f (x x))
```

Simply-typed fixed point for type T:

```
fixᵀ = λf:T→T.
  (λx:(μX.X→T).f (x x))
  (λx:(μX.X→T).f (x x))
```

Well-typed fixed point combinator

Untyped fixed point:

```
fix = λf.(λx.f (x x))(λx.f (x x))
```

Simply-typed fixed point for type T:

```
fixT = λf:T→T.
  (λx:(μX.X→T).f (x x))
  (λx:(μX.X→T).f (x x))
```

Corollary: recursive types break normalization property.

Two approaches to recursive types

How is the recursive type related to its one-step unfolding?

```
NatList vs <nil: Unit, cons: NatList>
```

Two approaches to recursive types

How is the recursive type related to its one-step unfolding?

```
NatList vs <nil: Unit, cons: NatList>
```

1. **Equi-recursive** approach says that those types are definitionally equal, as they stand for the same infinite tree.

Two approaches to recursive types

How is the recursive type related to its one-step unfolding?

```
NatList vs <nil: Unit, cons: NatList>
```

1. **Equi-recursive** approach says that those types are definitionally equal, as they stand for the same infinite tree.

2. **Iso-recursive** approach says that those types are distinct, but isomorphic (there explicit coercions between the two).

Two approaches to recursive types

How is the recursive type related to its one-step unfolding?

```
              unfold
NatList vs <nil: Unit, cons: NatList>
              fold
```

1. **Equi-recursive** approach says that those types are definitionally equal, as they stand for the same infinite tree.

2. **Iso-recursive** approach says that those types are distinct, but isomorphic (there explicit coercions between the two).

Two approaches to recursive types

How is the recursive type related to its one-step unfolding?

```
                    unfold
NatList vs <nil: Unit, cons: NatList>
                    fold
```

1. **Equi-recursive** is typically easier to reason from the programmer's perspective, but harder to implement.

2. **Iso-recursive** is easier to implement. Also, coercions (folding/unfolding) can be introduced by the typechecker.

Iso-recursive types

unfold

`NatList vs <nil: Unit, cons: NatList>`

fold

$$\begin{tikzcd} \mu X.T \arrow[rr, bend left, "{\mathbf{unfold}[\mu X.T]}"] & & {[X \mapsto \mu X.T]T} \arrow[ll, bend left, "{\mathbf{fold}[\mu X.T]}"] \end{tikzcd}$$

Iso-recursive types

`t ::= …`		***terms***
	`fold[T] t`	*folding*
	`unfold[T] t`	*unfolding*
`v ::= …`		***values***
	`fold[T] v`	*folding*
`T ::= …`		***types***
	`X`	*type variable*
	`μX.T`	*recursive type*

$$\mathbf{unfold}[S](\mathbf{fold}[T]\ v_1) \longrightarrow v_1$$

$$\frac{\Gamma \vdash t : [X \mapsto U]T \qquad U = \mu X.T}{\Gamma \vdash \mathbf{fold}[U]\ t : U}$$

$$\frac{\Gamma \vdash t : U \qquad U = \mu X.T}{\Gamma \vdash \mathbf{unfold}[U]\ t : [X \mapsto U]T}$$

Iso-recursive types: using coercions implicitly

```
NatList = μX. <nil: Unit, cons: {Nat, X}>
NLBody = <nil: Unit, cons: NatList>
```

Iso-recursive types: using coercions implicitly

```
NatList = μX. <nil: Unit, cons: {Nat, X}>

NLBody = <nil: Unit, cons: NatList>

let nil : NatList =
      fold[NatList] (<nil = unit> as NLBody)
```

Iso-recursive types: using coercions implicitly

```
NatList = μX. <nil: Unit, cons: {Nat, X}>

NLBody = <nil: Unit, cons: NatList>

let nil : NatList =
      fold[NatList] (<nil = unit> as NLBody)

let cons : Nat → NatList → NatList =
      λn:Nat.λl:NatList.
            fold[NatList] (<cons = {n, l}> as NLBody)
```

Iso-recursive types: using coercions implicitly

```
NatList = μX. <nil: Unit, cons: {Nat, X}>

NLBody = <nil: Unit, cons: NatList>

let nil : NatList =
      fold[NatList] (<nil = unit> as NLBody)

let cons : Nat → NatList → NatList =
      λn:Nat.λl:NatList.
            fold[NatList] (<cons = {n, l}> as NLBody)

let isnil : NatList → Bool =
      λl:NatList. case unfold[NatList] l of
         <nil  = _>        ⇒ true
        | <cons = {_, _}> ⇒ false
```

Recursive types and subtyping

```
Even <: Nat
```

What should be the relationship between the following types?

```
μX.Nat→(Even×X)          and
          μX.Even→(Nat×X)
```

Subtyping of iso-recursive types

$$\frac{\Sigma,\ X <: Y \vdash S <: T}{\Sigma \vdash \mu X.S <: \mu Y.T}$$

$$\Sigma,\ X <: Y \vdash X <: Y$$

Generating functions

Definition 10.4. A function $F \in P(U) \to P(U)$ is ***monotone*** if $X \subseteq Y$ implies $F(X) \subseteq F(Y)$.

Generating functions

Definition 10.4. A function $F \in P(U) \to P(U)$ is ***monotone*** if $X \subseteq Y$ implies $F(X) \subseteq F(Y)$.

Definition 10.5. Let X be a subset of U.

1. X is ***F-closed*** if $F(X) \subseteq X$.
2. X is ***F-consistent*** if $X \subseteq F(X)$.
3. X is a **fixed point of F** if $X = F(X)$.

Generating functions: example

$G(\varnothing) = \{c\}$

$G(\{a\}) = \{c\}$
$G(\{b\}) = \{c\}$
$G(\{c\}) = \{b, c\}$

$G(\{a,b\}) = \{c\}$
$G(\{a,c\}) = \{b,c\}$
$G(\{b,c\}) = \{a,b,c\}$

$G(\{a,b,c\}) = \{a,b,c\}$

G-closed sets

G-consistent sets

Generating functions: example

G(ø) = {c}

G({a}) = {c}
G({b}) = {c}
G({c}) = {b, c}

G({a,b}) = {c}
G({a,c}) = {b,c}
G({b,c}) = {a,b,c}

G({a,b,c}) = {a,b,c}

G-closed sets

{a,b,c}

G-consistent sets

ø

{c}

{b,c}

{a,b,c}

Least and greatest fixpoints

Theorem 10.6.

1. The intersection of all F-closed sets
   is the least fixed point of F. (we will write μF)
2. The union of all F-consistent sets
   is the greatest fixed point of F. (we will write νF)

Least and greatest fixpoints

Theorem 10.6.

1. The intersection of all F-closed sets
   is the least fixed point of F. (we will write μF)
2. The union of all F-consistent sets
   is the greatest fixed point of F. (we will write νF)

Corollary.

1. ***Principle of induction.*** If X is F-closed, then $\mu F \subseteq X$.
2. ***Principle of coinduction.*** If X is F-consistent, then $X \subseteq \nu F$.

Summary

- ❏ Structural recursive types
- ❏ Equi-recursive vs Iso-recursive
- ❏ Formal definitions for iso-recursive types
- ❏ Recursive types and subtyping
- ❏ Least and greatest fixed points

See you next time!