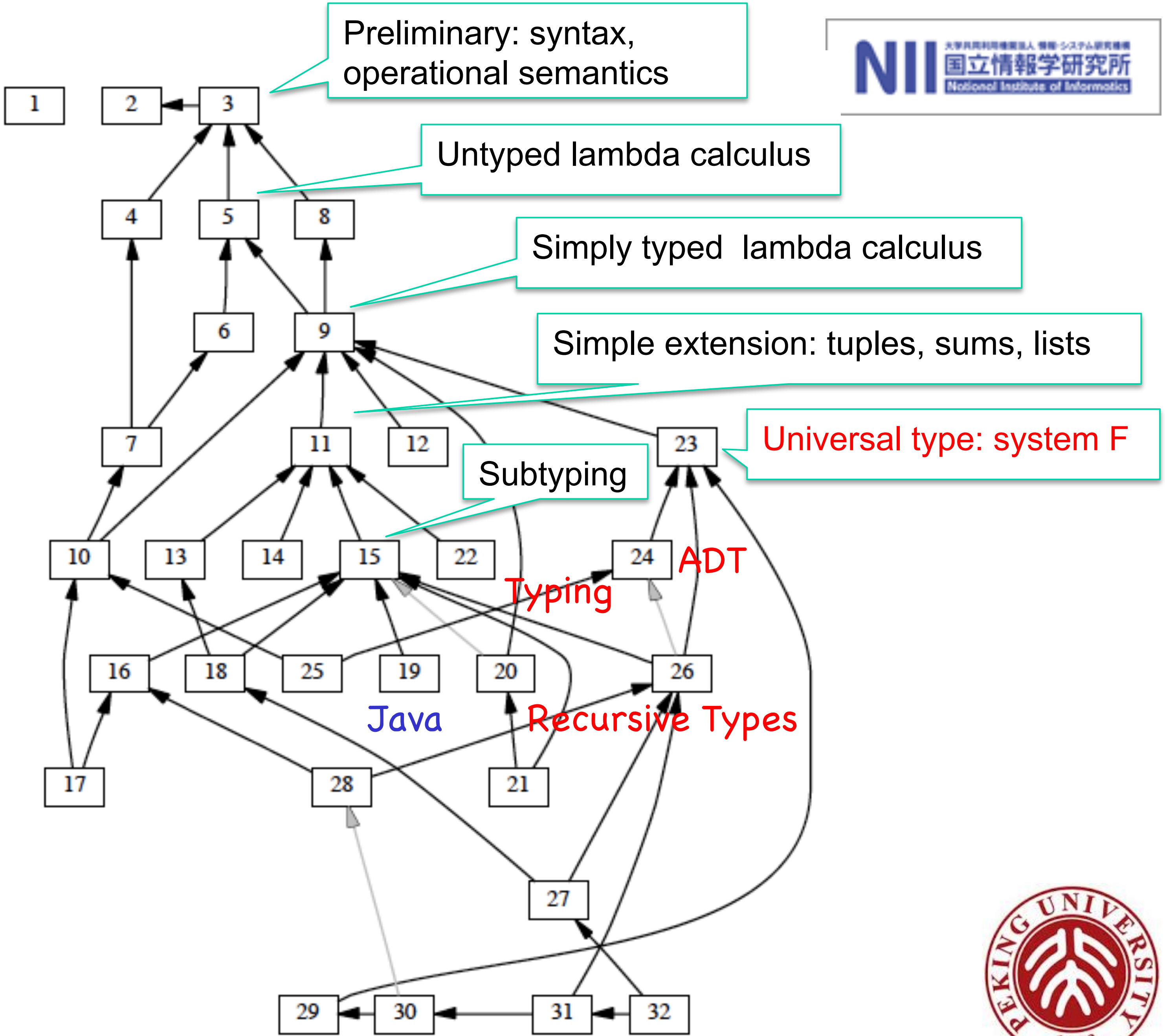
Review


Preliminary: syntax, operational semantics
Untyped lambda calculus
Simply typed lambda calculus
Simple extension: tuples, sums, lists
Universal type: system F
Subtyping
ADT
Typing
Java
Recursive Types
1
2
3
4
5
8
6
9
7
11
12
23
10
13
14
15
22
24
16
18
25
19
20
26
17
28
21
27
29
30
31
32

Chapter 20: Recursive Types

Examples

Formalities

Subtyping

Review: Lists Defined in Chapter 11

- List T describes finite-length lists whose elements are drawn from T.

→ 𝔹 List *Extends λ→ (9-1) with booleans (8-1)*

New syntactic forms

t ::=	...		*terms:*
	nil[T]		*empty list*
	cons[T] t t		*list constructor*
	isnil[T] t		*test for empty list*
	head[T] t		*head of a list*
	tail[T] t		*tail of a list*
v ::=	...		*values:*
	nil[T]		*empty list*
	cons[T] v v		*list constructor*
T ::=	...		*types:*
	List T		*type of lists*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{cons[T]}\ t_1\ t_2 \longrightarrow \texttt{cons[T]}\ t_1'\ t_2} \quad \text{(E-Cons1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\texttt{cons[T]}\ v_1\ t_2 \longrightarrow \texttt{cons[T]}\ v_1\ t_2'} \quad \text{(E-Cons2)}$$

$$\texttt{isnil[S] (nil[T])} \longrightarrow \texttt{true} \quad \text{(E-IsnilNil)}$$

$$\texttt{isnil[S] (cons[T]}\ v_1\ v_2\texttt{)} \longrightarrow \texttt{false} \quad \text{(E-IsnilCons)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{isnil[T]}\ t_1 \longrightarrow \texttt{isnil[T]}\ t_1'} \quad \text{(E-Isnil)}$$

$$\texttt{head[S] (cons[T]}\ v_1\ v_2\texttt{)} \longrightarrow v_1 \quad \text{(E-HeadCons)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{head[T]}\ t_1 \longrightarrow \texttt{head[T]}\ t_1'} \quad \text{(E-Head)}$$

$$\texttt{tail[S] (cons[T]}\ v_1\ v_2\texttt{)} \longrightarrow v_2 \quad \text{(E-TailCons)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{tail[T]}\ t_1 \longrightarrow \texttt{tail[T]}\ t_1'} \quad \text{(E-Tail)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \texttt{nil}\ [T_1] : \texttt{List}\ T_1 \quad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : \texttt{List}\ T_1}{\Gamma \vdash \texttt{cons}[T_1]\ t_1\ t_2 : \texttt{List}\ T_1} \quad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{isnil}[T_{11}]\ t_1 : \texttt{Bool}} \quad \text{(T-Isnil)}$$

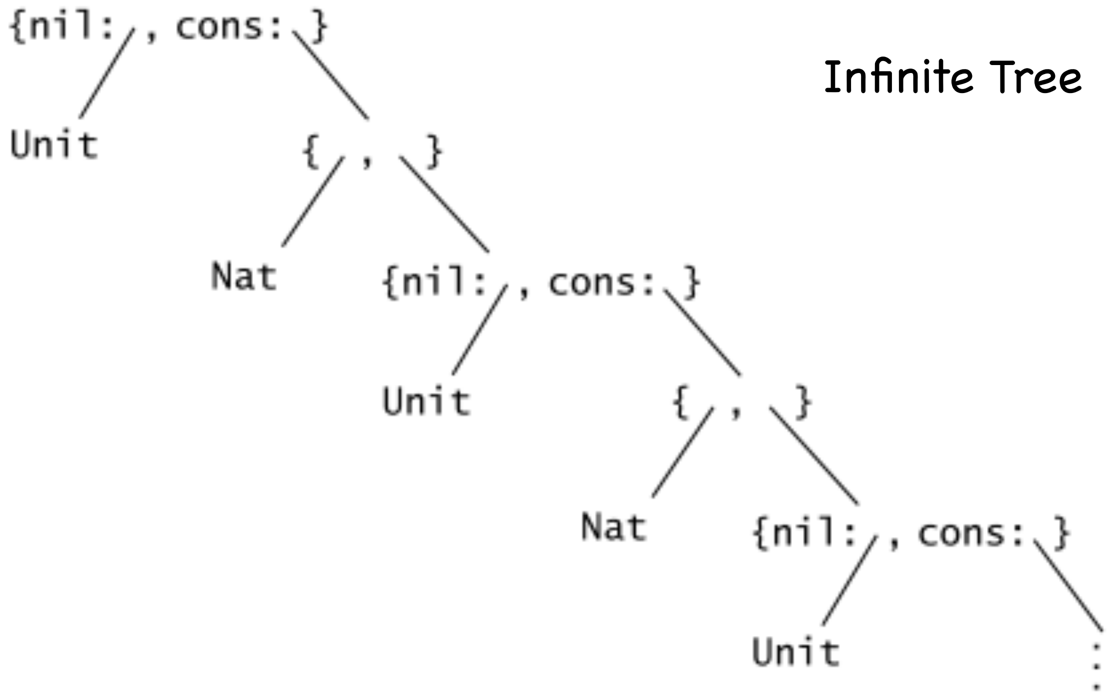
$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{head}[T_{11}]\ t_1 : T_{11}} \quad \text{(T-Head)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{tail}[T_{11}]\ t_1 : \texttt{List}\ T_{11}} \quad \text{(T-Tail)}$$

Examples of Recursive Types

Lists

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



Infinite Tree

$$NatList = \mu X.\ \langle nil:Unit,\ cons:\{Nat,X\}\rangle$$

This means that let NatList be the infinite type satisfying the equation:

$$X = \langle nil:Unit,\ cons:\{Nat,\ X\}\rangle.$$

Defining functions over lists

- nil = <nil=unit> as NatList
- cons = λn:Nat. λl:NatList. <cons={n,l}> as NatList
- isnil = λl:NatList. case l of
  <nil=u> ⇒ true
  | <cons=p> ⇒ false;
- hd = λl:NatList. case l of <nil=u> ⇒ 0 | <cons=p> ⇒ p.1
- tl = λl:NatList. case l of <nil=u> ⇒ l | <cons=p> ⇒ p.2
- sumlist = fix (λs:NatList→Nat. λl:NatList.
  if isnil l then 0 else plus (hd l) (s (tl l)))

Hungry Functions

- **Hungry Functions**: accepting any number of numeric arguments and always return a new function that is hungry for more

$$\text{Hungry} = \mu A.\ \text{Nat} \to A$$

f : Hungry

f = fix (λ f: Nat→Hungry. λ n:Nat. f)

f 0 1 2 3 4 5 : Hugary

Streams

- **Streams**: consuming an arbitrary number of unit values, each time returning a pair of a number and a new stream

$$\text{Stream} = \mu A.\ \text{Unit} \to \{\text{Nat}, A\};$$

upfrom0 : Stream
upfrom0 = fix (λ f: Nat→Stream. λ n:Nat. λ _:Unit.
{n,f (succ n)}) 0;

hd : Stream → Nat
hd = λ s:Stream. (s unit).1

(Process = μA. Nat→ {Nat, A})

Objects

- **Objects**

Counter = μC. { get : Nat,
inc : Unit→C,
dec : Unit→C }

c : Counter

```
c = let create = fix (λ f: {x:Nat}→Counter.  λ s: {x:Nat}.
                      { get = s.x,
                        inc = λ _:Unit. f {x=succ(s.x)},
                        dec = λ _:Unit. f {x=pred(s.x)} })
    in create {x=0};
```

((c.inc unit).inc unit).get ➔ 2

Recursive Values from Recursive Types

- **Recursive Values from Recursive Types**

$F = \mu A.A \to T$

$\text{fixT} = \lambda f{:}T \to T.\ (\lambda x{:}(\mu A.A \to T).\ f\ (x\ x))$
$\qquad\qquad\qquad (\lambda x{:}(\mu A.A \to T).\ f\ (x\ x))$

(Breaking the strong normalizing property:
diverge = λ_:Unit. fixT (λx:T. x) becomes typable)

Untyped Lambda Calculus

- **Untyped Lambda-Calculus**: we can embed the whole untyped lambda-calculus - in a well-typed way - into a statically typed language with recursive types.

$$D = \mu X. X \to X;$$

lam : D

lam = λ f:D→D. f as D;

ap : D

ap = λ f:D. λ a:D. f a;

Formalities

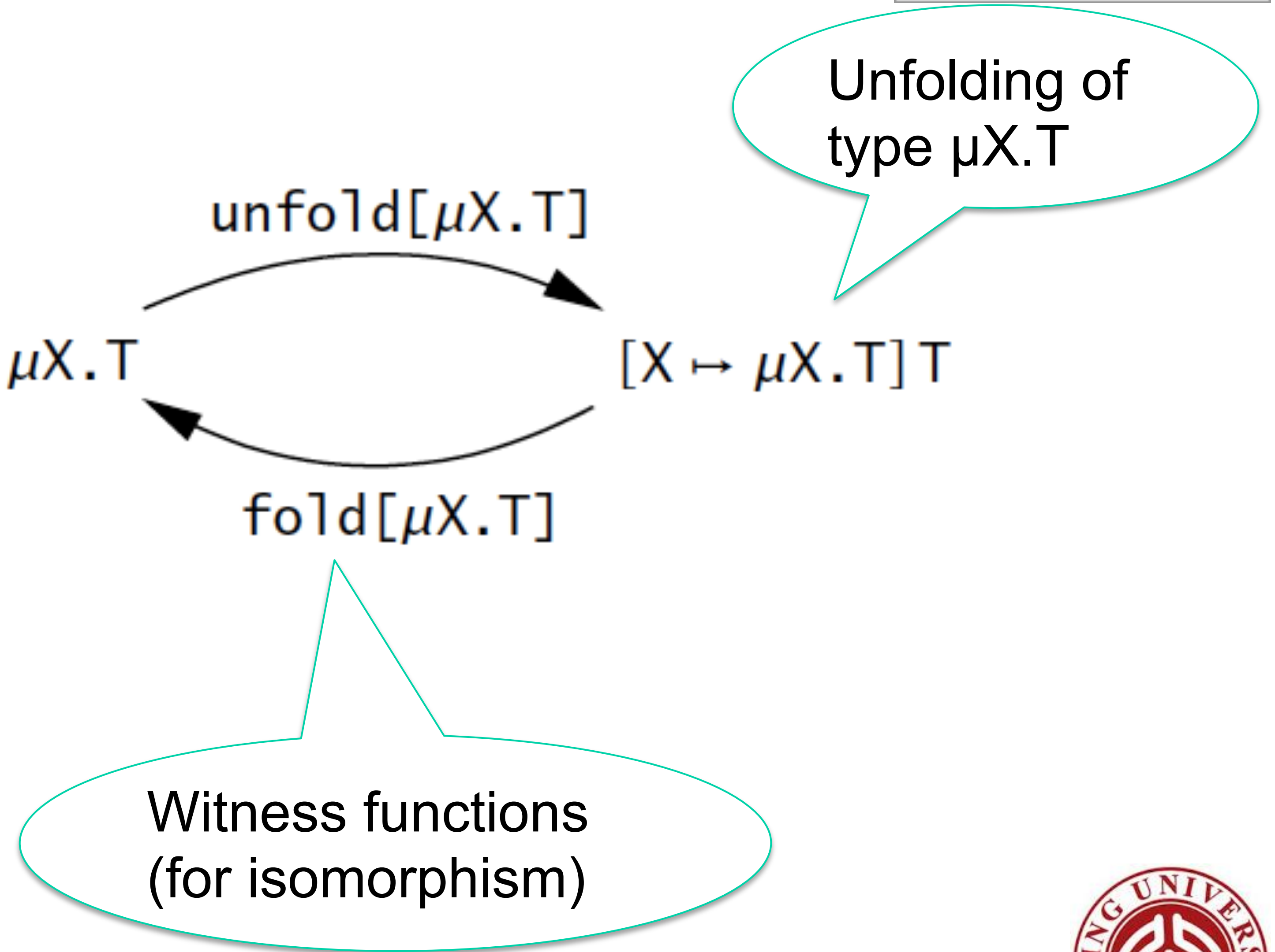
What is the relation between the type $\mu X.T$ and its one-step unfolding?

Two Approaches

- The equi-recursive approach
  - takes these two type expressions as definitionally equal— interchangeable in all contexts— since they stand for the same infinite tree.
  - more intuitive, but places stronger demands on the typechecker.

- 2. The iso-recursive approach
  - takes a recursive type and its unfolding as different, but isomorphic.
  - Notationally heavier, requiring programs to be decorated with fold and unfold instructions wherever recursive types are used.

The Iso-Recursive Approach



Iso-recursive types (λμ)

$\rightarrow$ μ *Extends* $\lambda_{\rightarrow}$ *(9-1)*

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*

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{unfold [S] (fold [T] } \texttt{v}_1) \longrightarrow \texttt{v}_1 \quad \text{(E-UnfldFld)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{fold [T] } \texttt{t}_1 \longrightarrow \texttt{fold [T] } \texttt{t}_1'} \quad \text{(E-Fld)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{unfold [T] } \texttt{t}_1 \longrightarrow \texttt{unfold [T] } \texttt{t}_1'} \quad \text{(E-Unfld)}$$

New typing rules $\boxed{\Gamma \vdash \texttt{t} : \texttt{T}}$

$$\frac{\texttt{U} = \mu \texttt{X.T}_1 \qquad \Gamma \vdash \texttt{t}_1 : [\texttt{X} \mapsto \texttt{U}]\texttt{T}_1}{\Gamma \vdash \texttt{fold [U] } \texttt{t}_1 : \texttt{U}} \quad \text{(T-Fld)}$$

$$\frac{\texttt{U} = \mu \texttt{X.T}_1 \qquad \Gamma \vdash \texttt{t}_1 : \texttt{U}}{\Gamma \vdash \texttt{unfold [U] } \texttt{t}_1 : [\texttt{X} \mapsto \texttt{U}]\texttt{T}_1} \quad \text{(T-Unfld)}$$

Lists (Revisited)

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

- 1-step unfolding of NatList:

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

- Definitions of functions on NatList
  - Constructors
    - nil = fold [NatList] (<nil=unit> as NLBody)
    - Cons = λ n:Nat. λ l:NatList.
      fold [NatList] <cons={n,l}> as NLBody
  - Destructors
    - hd = λ l:NatList.
      case unfold [NatList] l of
      <nil=u> ⇒ 0
      | <cons=p> ⇒ p.1

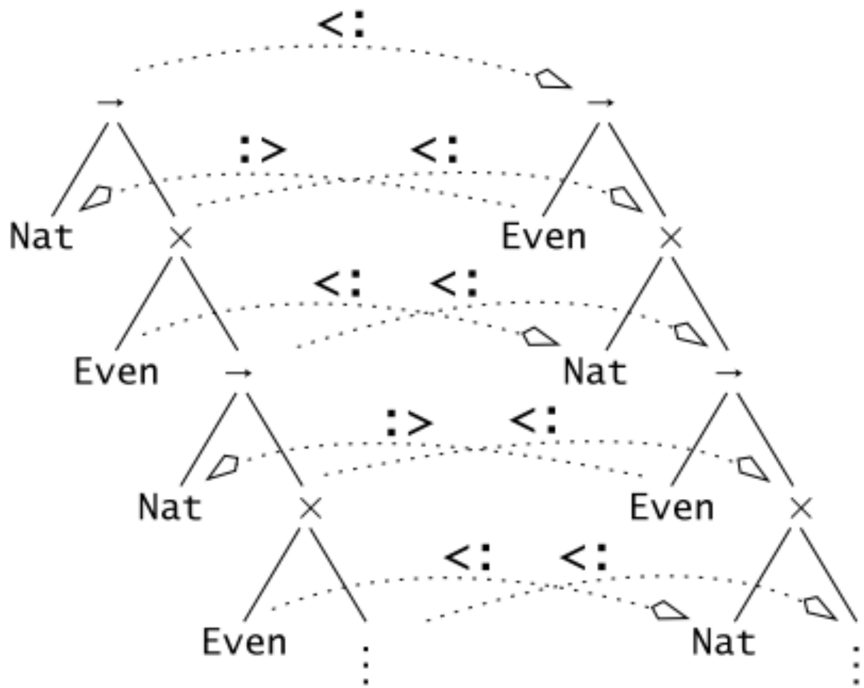
[Exercises: Define tl, sinil]

Subtyping

- Can we deduce

  $\mu X.\ Nat \rightarrow (Even \times X) <: \mu X.\ Even \rightarrow (Nat \times X)$

  from Even <: Nat?



Homework

Problem (Chapter 20)

Natural number can be defined recursively by

$$Nat = \mu X. \langle zero: Nil, succ: X \rangle$$

Define the following functions in terms of fold and unfold.

(1) isZero n: check whether a natural number n is. zero or not.

(2) add1 n: increase a natural number n by 1.

(3) plus m n: add two natural numbers.