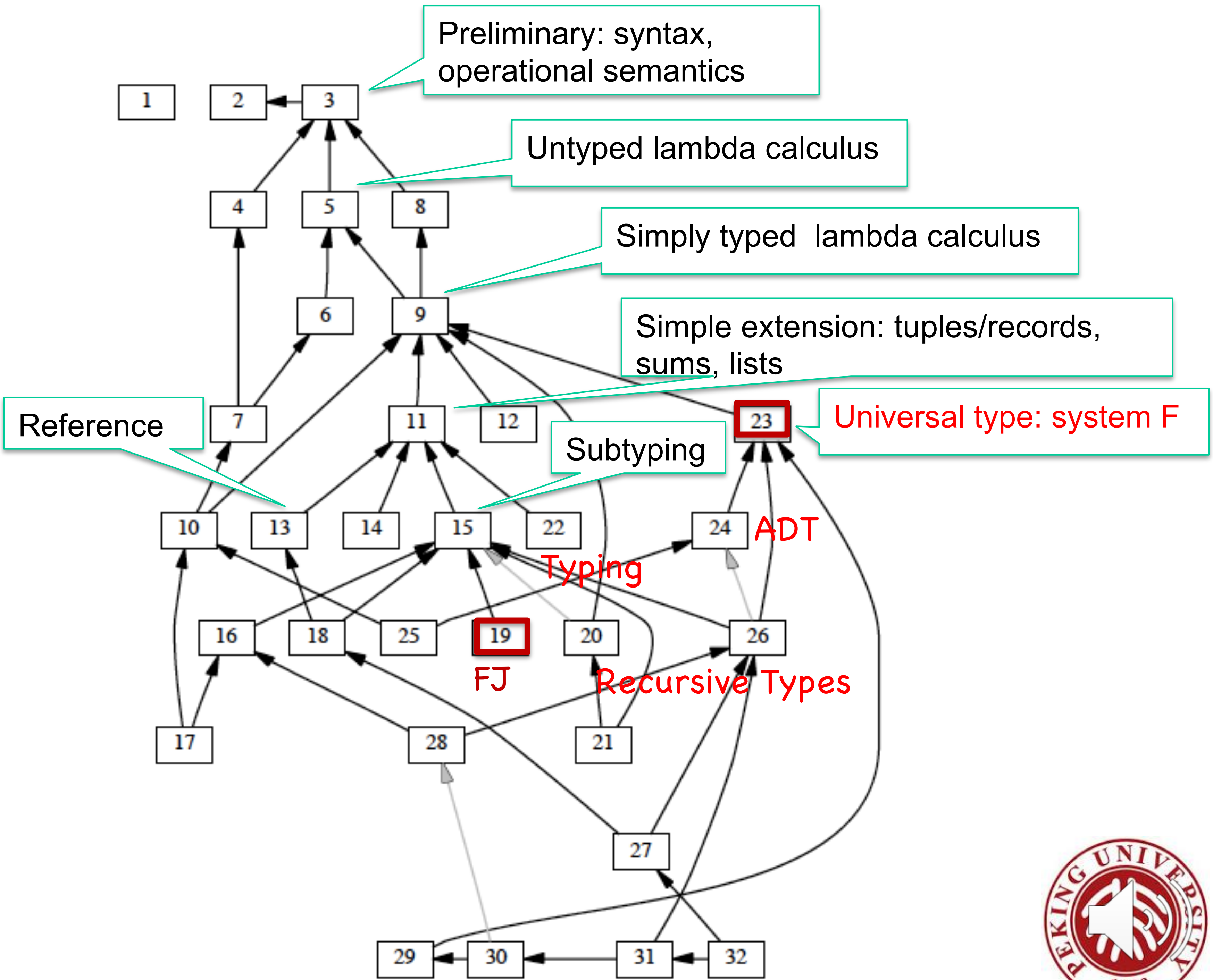
Chapter 20: Recursive Types

Examples

Formalities

Subtyping



Preliminary: syntax, operational semantics
Untyped lambda calculus
Simply typed lambda calculus
Simple extension: tuples/records, sums, lists
Universal type: system F
Reference
Subtyping
ADT
Typing
FJ
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

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 $\boxed{t \longrightarrow t'}$

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

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

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

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

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

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

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

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

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

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

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

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

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

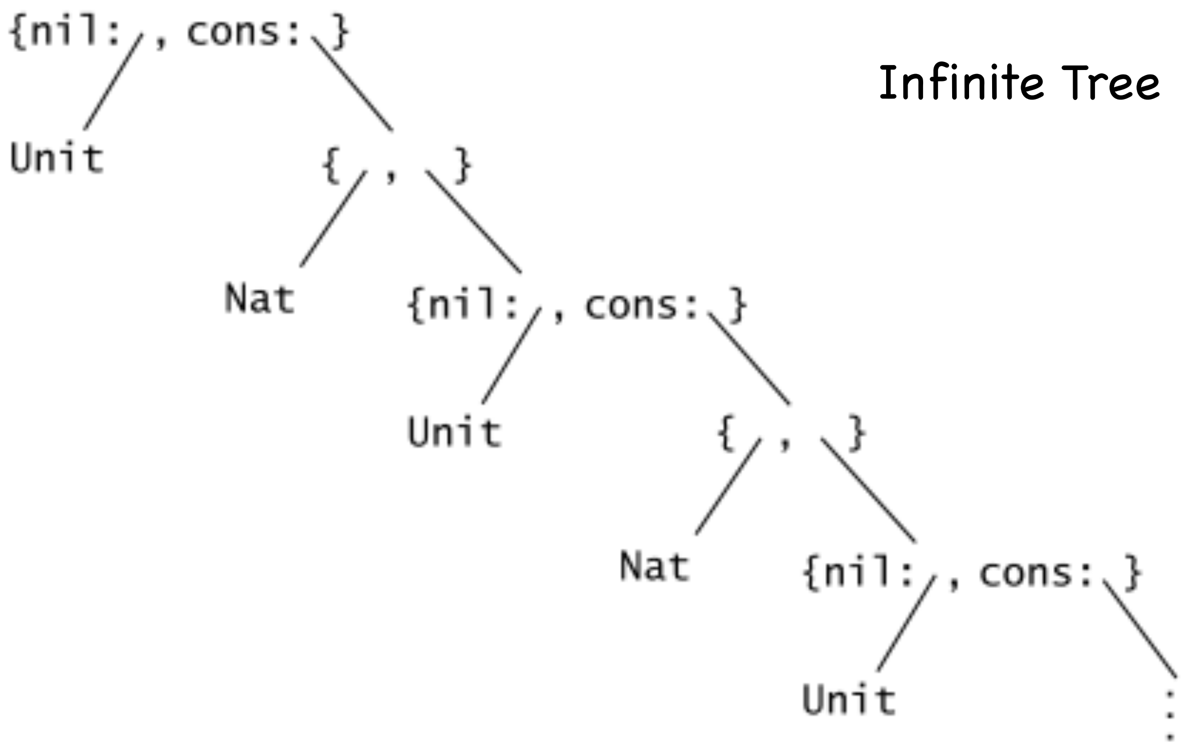
$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{head}[T_{11}]\ t_1 : T_{11}} \quad \text{(T-Head)}$$

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

Examples of Recursive Types

Lists

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



Infinite Tree

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

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

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

List

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

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 (λf: Nat$\to$Hungry. λn:Nat. f)

f 0 1 2 3 4 5 : Hungary

Streams

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

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

hd : Stream → Nat

hd = λs:Stream. (s unit).1

upfrom0 : Stream

upfrom0 = fix (λf: Nat→Stream. λn:Nat. λ_:Unit.
{n,f (succ n)}) 0;

(Process = μA. Nat→ {Nat, A})

20.1.2 EXERCISE [RECOMMENDED, ★★]: Define a stream that yields successive elements of the Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, ...). □

```
fib = fix (λf: Nat→Nat→Stream.
                λm:Nat. λn:Nat.
                    λ_:Unit. {n, f n (plus m n)})
        0 1;
```

Objects

- **Objects**

```
Counter = μC. { get : Nat,
                inc : Unit→C,
                dec : Unit→C }
c : Counter
c = let create = fix (λf: {x:Nat}→Counter. λs: {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**

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

$fixT = \lambda f:T \rightarrow T.\ (\lambda x:(\mu A.A \rightarrow T).\ f\ (x\ x))$
$\qquad\qquad\qquad\qquad (\lambda x:(\mu A.A \rightarrow T).\ f\ (x\ x))$

(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 = λf:D$\to$D. f as D;

ap : D

ap = λf:D. λa:D. f a;

- Embedding

$$
\begin{array}{lcl}
\mathtt{x}^{\star} & = & \mathtt{x} \\
(\lambda\mathtt{x}.\mathtt{M})^{\star} & = & \mathtt{lam}\,(\lambda\mathtt{x}{:}\mathtt{D}.\ \mathtt{M}^{\star}) \\
(\mathtt{M}\ \mathtt{N})^{\star} & = & \mathtt{ap}\ \mathtt{M}^{\star}\ \mathtt{N}^{\star}
\end{array}
$$

Formalities

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

NatList $\sim$ <nil:Unit,cons:{Nat,NatList}>

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 type-checker.
- 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

Unfolding of type μX.T

$$\begin{tikzcd} \mu X.T \arrow[r, bend left, "\mathtt{unfold}[\mu X.T]"] & {[X \mapsto \mu X.T]T} \arrow[l, bend left, "\mathtt{fold}[\mu X.T]"] \end{tikzcd}$$

Witness functions (for isomorphism)

Q: What is the 1-step unfolding of μX.<nil:Unit,cons:{Nat,X}>?

Iso-recursive types (λμ)

→ μ *Extends $\lambda_{\to}$ (9-1)*

$\mathtt{t}$	$::=$	...	*terms:*
		$\mathtt{fold}\ [\mathtt{T}]\ \mathtt{t}$	*folding*
		$\mathtt{unfold}\ [\mathtt{T}]\ \mathtt{t}$	*unfolding*
$\mathtt{v}$	$::=$	...	*values:*
		$\mathtt{fold}\ [\mathtt{T}]\ \mathtt{v}$	*folding*
$\mathtt{T}$	$::=$	...	*types:*
		$\mathtt{X}$	*type variable*
		$\mu\mathtt{X}.\mathtt{T}$	*recursive type*

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

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

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

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

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

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

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

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 = λn:Nat. λl:NatList.
fold [NatList] <cons={n,l}> as NLBody
  - Destructors
    - hd = λl:NatList.
case unfold [NatList] l of
<nil=u> ⇒ 0
| <cons=p> ⇒ p.1

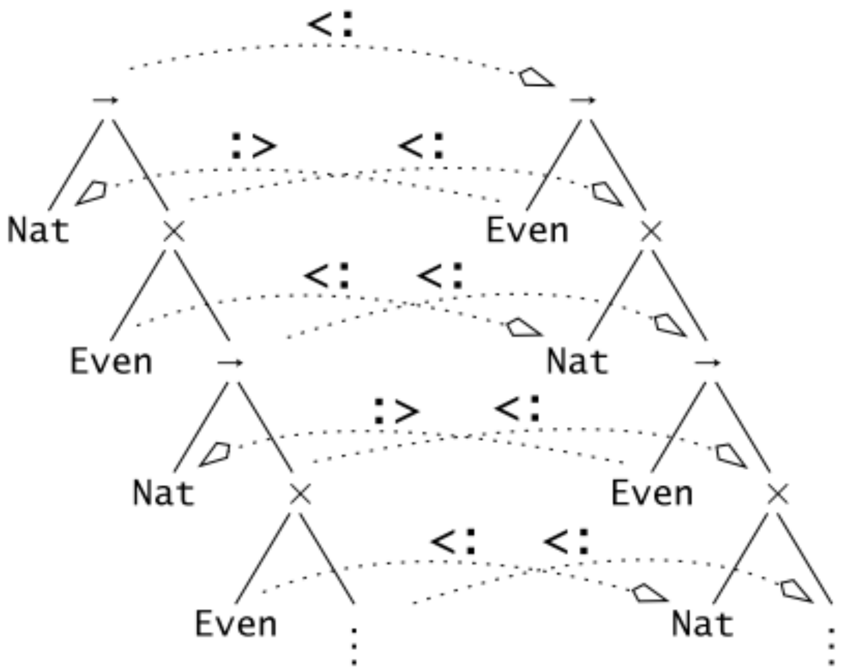
[Exercises: Define tl, isnil]

Subtyping

- Can we deduce

$$\mu X.\ \mathrm{Nat} \to (\mathrm{Even} \times X) <: \mu X.\ \mathrm{Even} \to (\mathrm{Nat} \times X)$$

from Even <: Nat?



infinite subtyping derivations over infinite ypes.

Homework

Problem (Chapter 20)

Natural number can be defined recursively by

Nat = μX. <zero: Nil, succ: X>

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.