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 $\boxed{\Gamma \vdash t : T}$

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

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

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

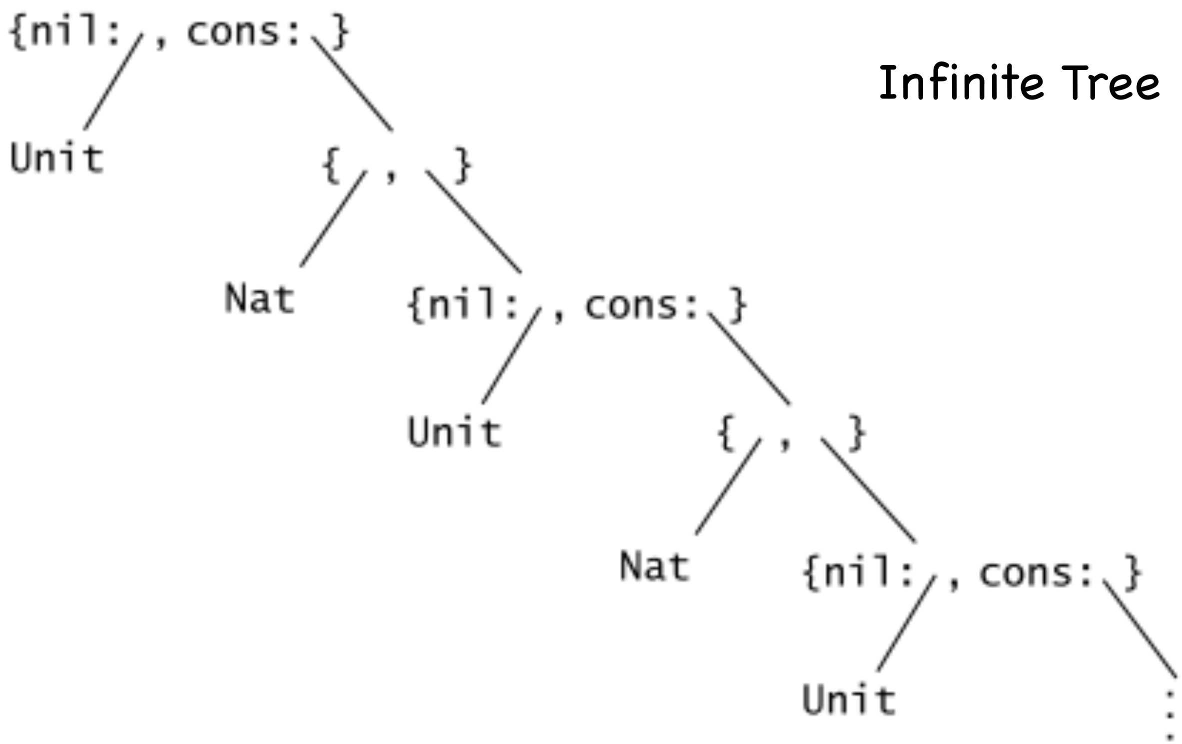
$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{head}[T_{11}]\ t_1 : T_{11}} \quad \text{(T-Head)}$$

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

Examples of Recursive Types

Lists

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



Infinite Tree

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

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

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

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

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

f : Hungry

f = fix (λf: Nat→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 = μA.A→T

  fixT = λf:T→T. (λx:(μA.A→T). f (x x))
  (λx:(μA.A→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

$\text{lam} = \lambda f{:}D \to D.\ f \text{ as } D;$

ap : D

$\text{ap} = \lambda f{:}D.\ \lambda a{:}D.\ f\ a;$

- Embedding

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

Formalities

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

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

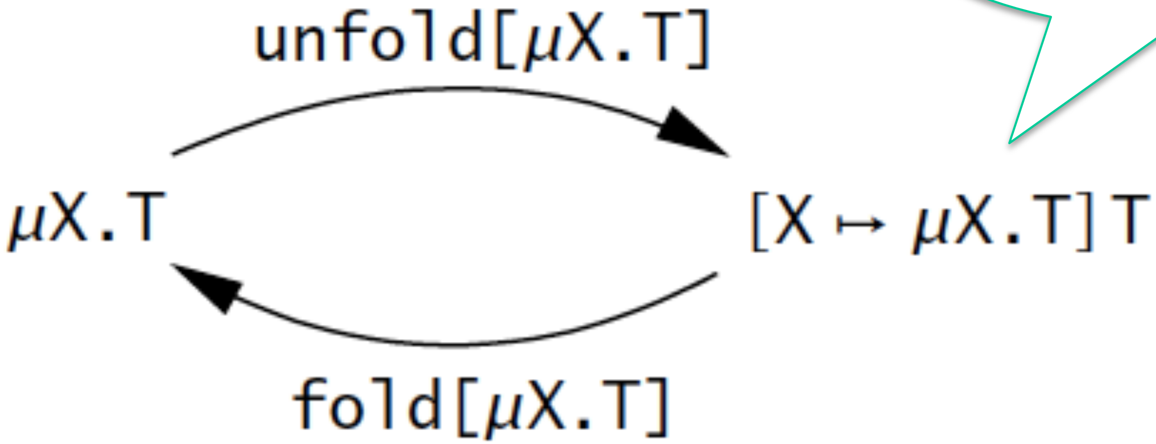
Two Approaches

- 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



Witness functions (for isomorphism)

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

Iso-recursive types (λμ)

→ μ *Extends λ→ (9-1)*

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

New evaluation rules $t \longrightarrow t'$

$$\texttt{unfold [S] (fold [T] } v_1\texttt{)} \longrightarrow v_1 \quad \text{(E-UnfldFld)}$$

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

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

New typing rules $\Gamma \vdash t : T$

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : [X \mapsto U]T_1}{\Gamma \vdash \texttt{fold [U] } t_1 : U} \quad \text{(T-Fld)}$$

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : U}{\Gamma \vdash \texttt{unfold [U] } t_1 : [X \mapsto U]T_1} \quad \text{(T-Unfld)}$$

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 = λn:Nat. λl:NatList.

      fold [NatList] <cons={n,l}> as NLBody
  - Destructors
    - hd = λl:NatList.

      case unfold [NatList] l of

      <nil=u> $\Rightarrow$ 0

      | <cons=p> $\Rightarrow$ p.1

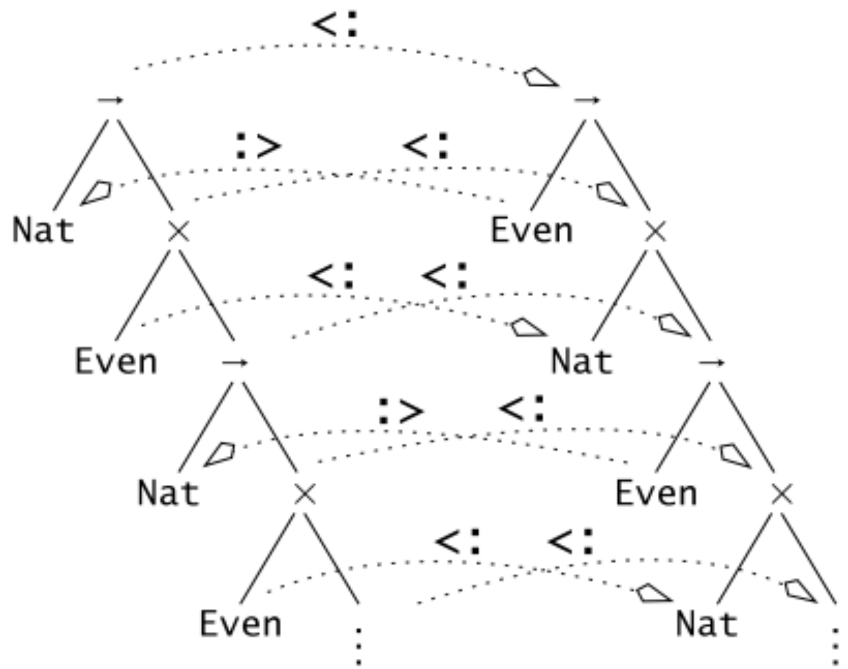
[Exercises: Define tl, isnil]

Subtyping

- Can we deduce

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

  from Even <: Nat?



infinite subtyping derivations over infinite ypes.

Homework

Problem (Chapter 20)

Natural number can be defined recursively by

$$Nat = \mu X. \text{<zero: Nil, succ: X>}$$

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.