编程语言的设计原理
Design Principles of Programming Languages

Haiyan Zhao, Di Wang
赵海燕，王迪

Peking University, Spring Term 2023

Chap 20: Recursive Types

Examples

Formalities

Inductive Types

Coinductive Types

Subtyping

Review: Lists Defined in Chapter 11

$\texttt{List}\ \mathsf{T}$ describes finite-length lists whose elements are of type T.

Syntactic Forms

$$
\begin{aligned}
t &::= \ldots \mid \texttt{nil}[T] \mid \texttt{cons}[T]\ t\ t \mid \texttt{isnil}[T]\ t \mid \texttt{head}[T]\ t \mid \texttt{tail}[T]\ t \\
v &::= \ldots \mid \texttt{nil}[T] \mid \texttt{cons}[T]\ v\ v \\
T &::= \ldots \mid \texttt{List}\ T
\end{aligned}
$$

Typing Rules

$$\frac{}{\Gamma \vdash \texttt{nil}[T_1] : \texttt{List}\ T_1}\ \text{T-NIL} \qquad \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : \texttt{List}\ T_1}{\Gamma \vdash \texttt{cons}[T_1]\ t_1\ t_2 : \texttt{List}\ T_1}\ \text{T-CONS}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{isnil}[T_{11}]\ t_1 : \texttt{Bool}}\ \text{T-ISNIL} \qquad \frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{head}[T_{11}]\ t_1 : T_{11}}\ \text{T-HEAD} \qquad \frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{tail}[T_{11}]\ t_1 : \texttt{List}\ T_{11}}\ \text{T-TAIL}$$

Examples of Recursive Types

Lists

Question

Can we define list types in simply-typed lambda-calculus with extensions?

Remark

We have studied **tuples** and **variants**.

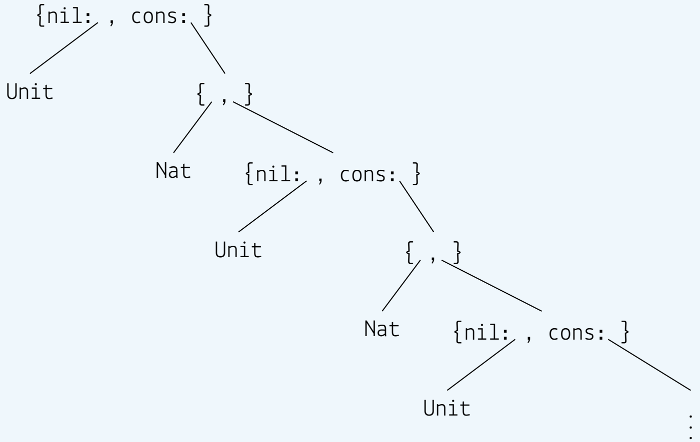
- Tuples: $\{T_i{}^{i \in 1...n}\}$
- Variants: $<l_i : T_i{}^{i \in 1...n}>$

Does the following definition work?

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

NatList as a Infinite Tree

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



Structural Recursive Types

Recursion Operator μ

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

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

$$X = \texttt{<nil : Unit, cons : \{Nat, X\}>}$$

Aside (Solving Type Equations)

Let $[\![T]\!]$ be the set of values of type T, e.g., $[\![\texttt{Unit}]\!] = \{\texttt{unit}\}$, $[\![\texttt{Nat}]\!] = \mathbb{N}$.
The solution $[\![X]\!]$ to the equation above should satisfy:

$$[\![X]\!] = \left\{\texttt{<nil=unit>}\right\} \cup \left\{\texttt{<cons=\{}v_1, v_2\texttt{\}>} \mid v_1 \in [\![\texttt{Nat}]\!], v_2 \in [\![X]\!]\right\}$$

Lists (cont.)

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

nil = <nil=unit> as NatList;
▶ nil : NatList
cons = λn:Nat. λl:NatList. <cons={n,l}> as NatList;
▶ cons : Nat → NatList → NatList

isnil = λl:NatList. case l of <nil=u> ⇒ true | <cons=p> ⇒ false;
▶ isnil : NatList → Bool
hd = λl:NatList. case l of <nil=u> ⇒ 0 | <cons=p> ⇒ p.1;
▶ hd : NatList → Nat
tl = λl:NatList. case l of <nil=u> ⇒ l | <cons=p> ⇒ p.2;
▶ tl : NatList → NatList

sumlist = fix (λs:NatList→Nat. λl:NatList.
                    if isnil l then 0 else plus (hd l) (s (tl l)));
▶ sumlist : NatList → Nat
```

Hungry Functions

Hungry Functions

A hungry function accepts any number of arguments and always return a new function that is hungry for more.

```
Hungry = μA. Nat→A;

f = fix (λf:Nat→Hungry. λn:Nat. f);
▶ f : Nat→Nat→Hungry

f 0 1 2 3 4 5;
▶ <fun> : Hungry
```

Streams

Streams

A stream consumes an arbitrary number of unit values, each time returning a pair of a value and a new stream.

```
Stream = μA. Unit→{Nat,A};

hd = λs:Stream. (s unit).1;
▶ hd : Stream → Nat
tl = λs:Stream. (s unit).2;
▶ tl : Stream → (μA. Unit→{Nat,A})

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

Question (Exercise 20.1.2)

Define a stream that yields successive elements of the Fibonacci sequence $(1, 1, 2, 3, 5, 8, 13, \ldots)$.

Streams (cont.)

```
fib = fix (λf:Nat→Nat→Stream. λa:Nat. λb:Nat. λ_:Unit. {a,f b (plus a b)}) 1 1;
▶ fib : Unit→{Nat,Stream};

hd fib;
▶ 1 : Nat
hd (tl (tl (tl fib)));
▶ 3 : Nat
hd (tl (tl (tl (tl (tl fib)))));
▶ 13 : Nat
```

Processes

A process accepts a value and returns a value and a new process.

```
Process = μA. Nat→{Nat,A}
```

Objects

Purely Functional Objects

An object accepts a message and returns a response to that message and **a new object** if mutated.

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

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}) as Counter;
▶ c : Counter

((c.inc unit).inc unit).get;
▶ 2 : Nat
```

Divergence

Remark

Recall omega from untyped lambda-calculus:

```
omega = (λx. x x) (λx. x x)
```

We have omega $\longrightarrow$ omega $\longrightarrow$ omega $\longrightarrow \ldots$, i.e., omega diverges.

Suppose we want to type $x : T_x \vdash x\,x : T$ for a given T. We obtain a type equation:

$$T_x = T_x \to T$$

Thus T_x can be defined as $\mu A.A \to T$.

Well-Typed Divergence

```
omegaT = (λx:(μA.A→T). x x) (λx:(μA.A→T). x x);
▶ omegaT : T
```

Recursive types break the strong-normalization property!

Recursion

Remark

Recall the Y operator from untyped lambda-calculus:

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

For any f, the operator satisfies $\mathsf{Y}\ f \longrightarrow^* f\ ((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))) =_\beta f\ (\mathsf{Y}\ f)$.

Question

Can we give Y a type using recursive types?

```
Y_T = λf:T→T. (λx:(μA.A→T). f (x x)) (λx:(μA.A→T). f (x x));
▶ Y_T : (T→T) → T
```

Question (Homework)

Implement Y_T in OCaml. Does it really work as a fixed-point operator? Why? How to make it work?
Show your solution is effective by using it to define a factorial function.

Untyped Lambda-Calculus

We can embed the whole untyped lambda-calculus into a statically typed language with recursive types.

```
D = μX.X→X;

lam = λf:D→D. f as D;
▶ lam : (D→D) → D
ap = λf:D. λa:D. (f a) as D;
▶ ap : D → D → D
```

Let M be a closed untyped lambda-term. We can embed M, written $M^\star$, as an element of D:

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

Formalities

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

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

Two Approaches

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

The Equi-Recursive Approach

- Take these two type expressions as definitionally equal—**interchangeable in all contexts**—since they stand for the same infinite tree.
- This approach is more intuitive, but places stronger demands on the type-checker.

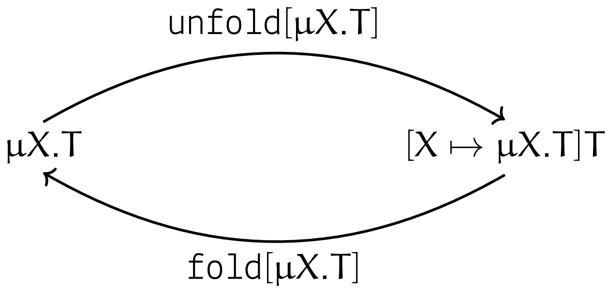
The Iso-Recursive Approach

- Take a recursive type and its unfolding as **different, but isomorphic**.
- This approach is notationally heavier, requiring programs to be decorated with `fold` and `unfold` instructions wherever recursive types are used.

Question

Which approach did we use in the previous examples?

The Iso-Recursive Approach



- $[X \mapsto \mu X.T]T$ is the one-step unfolding of $\mu X.T$.
- The pair of functions $\texttt{unfold}[\mu X.T]$ and $\texttt{fold}[\mu X.T]$ are witness functions for isomorphism.

Question

What is the one-step unfolding of $\mu X.\texttt{<nil : Unit, cons : \{Nat, X\}>}$?

Iso-Recursive Types ($\lambda\mu$)

Syntactic Forms

$$t ::= \ldots \mid \texttt{fold}\ [T]\ t \mid \texttt{unfold}\ [T]\ t \qquad v ::= \ldots \mid \texttt{fold}\ [T]\ v \qquad T ::= \ldots \mid X \mid \mu X.T$$

Evaluation Rules

(E-UnfldFld)

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

(E-Fld)

$$\frac{t_1 \longrightarrow t_1'}{\texttt{fold}\ [T]\ t_1 \longrightarrow \texttt{fold}\ [T]\ t_1'}$$

(E-Unfld)

$$\frac{t_1 \longrightarrow t_1'}{\texttt{unfold}\ [T]\ t_1 \longrightarrow \texttt{unfold}\ [T]\ t_1'}$$

Typing Rules

(T-Fld)

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : [X \mapsto U]T_1}{\Gamma \vdash \texttt{fold}\ [U]\ t_1 : U}$$

(T-Unfld)

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : U}{\Gamma \vdash \texttt{unfold}\ [U]\ t_1 : [X \mapsto U]T_1}$$

Lists (revisited)

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

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

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

hd = λl:NatList.
        case unfold [NatList] l of
          <nil=u> ⇒ 0
        | <cons=p> ⇒ p.1;
▶ hd : NatList → Nat
```

Question

OCaml is iso-recursive (by default). Where are the `fold`'s and `unfold`'s?

Inductive & Coinductive Types

Recursive Types are Useless as Logics

Remark (Curry-Howard Correspondence)

In simply-typed lambda-calculus, we can interpret types as logical propositions.

proposition $P \supset Q$	type $P \to Q$
proposition $P \wedge Q$	type $P \times Q$
proposition $P \vee Q$	type $P + Q$
proposition P is provable	type P is inhabited
proof of proposition P	term t of type P

Observation

Recursive types are so powerful that the strong-normalization property is broken.

```
omega_T = (λx:(μA.A→T). x x) (λx:(μA.A→T). x x);
▶ omega_T : T
```

The fact that $\mathtt{omega}_T$ is well-typed for every T means that **every proposition in the logic is provable**—that is, the logic is inconsistent.

Restricting Recursive Types

Question

What kinds of recursive types can ensure strong-normalization? What kinds cannot?

Lists	$\mu X.\texttt{<nil} : \texttt{Unit}, \texttt{cons} : \{\texttt{Nat}, X\}\texttt{>}$	✓
Streams	$\mu A.\texttt{Unit} \to \{\texttt{Nat}, A\}$	✓
Divergence	$\mu A.A \to \texttt{T}$	✗
Untyped lambda-calculus	$\mu X.X \to X$	✗

Observation

It seems problematic for a recursive type to recurse in the **contravariant** positions.

Inductive Types

$\mu X.T$ pos: "type $\mu X.T$ is positive"

$$\frac{}{\mu X.X \text{ pos}} \qquad \frac{}{\mu X.\mathtt{Unit} \text{ pos}} \qquad \frac{}{\mu X.\mathtt{Nat} \text{ pos}} \qquad \frac{\mu X.T_1 \text{ pos} \quad \mu X.T_2 \text{ pos}}{\mu X.T_1 \times T_2 \text{ pos}} \qquad \frac{\mu X.T_1 \text{ pos} \quad \mu X.T_2 \text{ pos}}{\mu X.T_1 + T_2 \text{ pos}}$$

$$\frac{T_1 \text{ type} \quad \mu X.T_2 \text{ pos}}{\mu X.T_1 \to T_2 \text{ pos}}$$

Question

Which of the following types are positive?

$$\mu X.\texttt{<nil}:\mathtt{Unit}, \mathtt{cons}:\{\mathtt{Nat}, X\}\texttt{>} \qquad \mu A.\mathtt{Unit} \to \{\mathtt{Nat}, A\} \qquad \mu A.A \to T \qquad \mu X.X \to X$$

Iterators for Well-Founded Recursion

Remark

Because of strong normalization, we cannot use the **fix** operator to define recursive functions on recursive types.

Principle

We can use **iteration** instead of general recursion. For $\mathsf{NatList} = \mu X.\texttt{<nil} : \mathsf{Unit}, \texttt{cons} : \{\mathsf{Nat}, X\}\texttt{>}$, we have

$$\frac{\Gamma \vdash t_1 : \mathsf{NatList} \qquad \Gamma, x : \texttt{<nil} : \mathsf{Unit}, \texttt{cons} : \{\mathsf{Nat}, S\}\texttt{>} \vdash t_2 : S}{\Gamma \vdash \textbf{iter}\ [\mathsf{NatList}]\ t_1\ \textbf{with}\ x.t_2 : S} \text{ T-ITER}$$

$$\frac{}{\textbf{iter}\ [\mathsf{NatList}]\ (\texttt{fold}\ [\mathsf{NatList}]\ \texttt{<nil=unit>})\ \textbf{with}\ x.t_2 \longrightarrow [x \mapsto \texttt{<nil=unit>}]t_2} \text{ E-ITER-NIL}$$

$$\frac{}{\begin{array}{c}\textbf{iter}\ [\mathsf{NatList}]\ (\texttt{fold}\ [\mathsf{NatList}]\ \texttt{<cons=}\{v_1, v_2\}\texttt{>})\ \textbf{with}\ x.t_2 \\ \longrightarrow \\ \textbf{let}\ y = (\textbf{iter}\ [\mathsf{NatList}]\ v_2\ \textbf{with}\ x.t_2)\ \textbf{in}\ [x \mapsto \texttt{<cons=}\{v_1, y\}\texttt{>}]t_2\end{array}} \text{ E-ITER-CONS}$$

Iterators for Well-Founded Recursion

```
sumlist = λl:NatList. iter [NatList] l
                        with x. case x of
                                 <nil=u> ⇒ 0
                               | <cons=p> ⇒ plus p.1 p.2;
▶ sumlist : NatList → Nat

append = λl1:NatList. λl2:NatList.
           iter [NatList] l1
             with x. case x of
                       <nil=u> ⇒ l2
                     | <cons=p> ⇒ fold [NatList] <cons={p.1,p.2}>;
▶ append : NatList → NatList → NatList
```

Streams (revisited)

Streams

A stream consumes an arbitrary number of unit values, each time returning a pair of a value and a new stream.

```
Stream = μA. Unit→{Nat,A};

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

Question

What is the difference between lists and streams?

Principle

Lists are defined as how to **construct** them.
Streams are defined as how to **destruct** them.

Coinductive Types

$\nu X.T$ pos: "type $\nu X.T$ is positive"

$$\frac{}{\nu X.X \text{ pos}} \qquad \frac{}{\nu X.\mathtt{Unit} \text{ pos}} \qquad \frac{}{\nu X.\mathtt{Nat} \text{ pos}} \qquad \frac{\nu X.T_1 \text{ pos} \qquad \nu X.T_2 \text{ pos}}{\nu X.T_1 \times T_2 \text{ pos}} \qquad \frac{\nu X.T_1 \text{ pos} \qquad \nu X.T_2 \text{ pos}}{\nu X.T_1 + T_2 \text{ pos}}$$

$$\frac{T_1 \text{ type} \qquad \nu X.T_2 \text{ pos}}{\nu X.T_1 \to T_2 \text{ pos}}$$

Remark (Solving Type Equations)

Let $[\![T]\!]$ be the set of values of type T, e.g., $[\![\mathtt{Unit}]\!] = \{\mathtt{unit}\}$, $[\![\mathtt{Nat}]\!] = \mathbb{N}$.
The solution $[\![X]\!]$ to the equation $X = \texttt{<nil : Unit, cons : \{Nat, X\}>}$ should satisfy:

$$[\![X]\!] = \left\{\texttt{<nil = unit>}\right\} \cup \left\{\texttt{<cons = }\{v_1, v_2\}\texttt{>} \mid v_1 \in [\![\mathtt{Nat}]\!], v_2 \in [\![X]\!]\right\}$$

Coinductive types are the greatest solutions. Inductive types are the least solutions.

Coinductive Types

Principle

We can use **generation** instead of general recursion or iteration. For $\texttt{Stream} = \nu X.\ \{\texttt{Nat}, X\}$, we have

$$\frac{\Gamma \vdash t_1 : S \qquad \Gamma, x : S \vdash t_2 : \{\mathsf{Nat}, S\}}{\Gamma \vdash \textbf{gen}\ [\texttt{Stream}]\ t_1\ \textbf{with}\ x.t_2 : \texttt{Stream}} \ \text{T-GEN}$$

$$\frac{}{\begin{array}{c}\texttt{unfold}\ [\texttt{Stream}]\ (\textbf{gen}\ [\texttt{Stream}]\ v_1\ \textbf{with}\ x.t_2) \\ \longrightarrow \\ \textbf{let}\ y = [x \mapsto v_1]t_2\ \textbf{in}\ \{y.1, (\textbf{gen}\ [\texttt{Stream}]\ y.2\ \textbf{with}\ x.t_2)\}\end{array}} \ \text{E-UNFOLD-GEN}$$

```
upfrom0 = gen [Stream] 0 with x. {x,succ(x)};
▶ upfrom0 : Stream
fib = gen [Stream] {1,1} with x. {x.1,{x.2,(plus x.1 x.2)}};
▶ fib : Stream
```

What's More

Summary

$$t ::= \ldots \mid \texttt{fold}\ [\texttt{NatList}]\ t \mid \textbf{iter}\ [\texttt{NatList}]\ t_1\ \textbf{with}\ x.t_2 \mid \texttt{unfold}\ [\texttt{Stream}]\ t \mid \textbf{gen}\ [\texttt{Stream}]\ t_1\ \textbf{with}\ x.t_2$$

$$v ::= \ldots \mid \texttt{fold}\ [\texttt{NatList}]\ v \mid \textbf{gen}\ [\texttt{Stream}]\ v_1\ \textbf{with}\ x.t_2$$

Aside

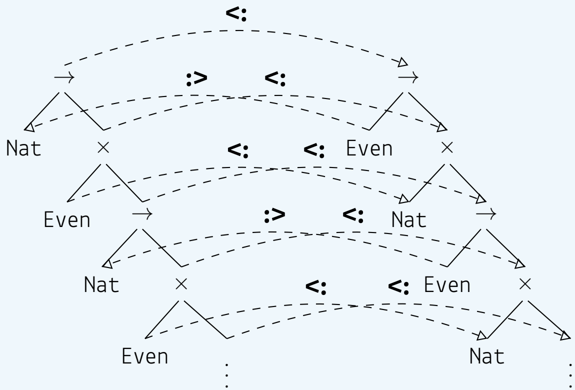
We only introduce the evaluation and typing rules for `NatList` and `Stream`.
How to evaluate and type-check general inductive types $\mu X.T$ and coinductive types $\nu X.T$?
How to prove the strong-normalization property?

Read more about inductive & coinductive types: N. P. Mendler. 1987. Recursive Types and Type Constraints in Second-Order Lambda Calculus. In *Logic in Computer Science* (LICS'87), 30–36.

Subtyping

Can we deduce the relation below, given that $\mathsf{Even} <: \mathsf{Nat}$?

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



Homework

Question

- Implement Y_T (shown on Slide 14) in OCaml. Does it really work as a fixed-point operator? Why?
- How to make it work? Show your solution is effective by using it to define a factorial function.
- Reformulate your solution with explicit `fold`'s and `unfold`'s. You may check your solution using the `fullisorec` checker.