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

Haiyan Zhao, Di Wang
赵海燕，王迪

Peking University, Spring Term 2025

Recursive Types
递归类型

Review: Lists Defined in Chapter 11

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

Syntactic Forms

$$t ::= \ldots \mid \texttt{nil}[T] \mid \texttt{cons}[T]\ t\ t \mid \texttt{isnil}[T]\ t \mid \texttt{head}[T]\ t \mid \texttt{tail}[T]\ t$$

$$v ::= \ldots \mid \texttt{nil}[T] \mid \texttt{cons}[T]\ v\ v$$

$$T ::= \ldots \mid \texttt{List}\ T$$

Typing Rules

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

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

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

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

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

BoolList: A Specialized Version

BoolList describes finite-length lists whose elements are of Booleans.

Syntactic Forms

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

Typing Rules

$$\frac{}{\Gamma \vdash \texttt{nil} : \texttt{BoolList}}\ \text{T-Nil} \qquad \frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_2 : \texttt{BoolList}}{\Gamma \vdash \texttt{cons}\ t_1\ t_2 : \texttt{BoolList}}\ \text{T-Cons}$$

$$\frac{\Gamma \vdash t_1 : \texttt{BoolList}}{\Gamma \vdash \texttt{isnil}\ t_1 : \texttt{Bool}}\ \text{T-IsNil} \qquad \frac{\Gamma \vdash t_1 : \texttt{BoolList}}{\Gamma \vdash \texttt{head}\ t_1 : \texttt{Bool}}\ \text{T-Head} \qquad \frac{\Gamma \vdash t_1 : \texttt{BoolList}}{\Gamma \vdash \texttt{tail}\ t_1 : \texttt{BoolList}}\ \text{T-Tail}$$

Review: Natural Numbers Defined in Chapter 8

$\mathtt{Nat}$ describes natural numbers.

Syntactic Forms

$$
\begin{aligned}
t &::= \ldots \mid \mathtt{0} \mid \mathtt{succ}\ t \mid \mathtt{iszero}\ t \mid \mathtt{pred}\ t \\
v &::= \ldots \mid \mathtt{0} \mid \mathtt{succ}\ v \\
T &::= \ldots \mid \mathtt{Nat}
\end{aligned}
$$

Typing Rules

$$\frac{}{\Gamma \vdash \mathtt{0} : \mathtt{Nat}}\ \text{T-Zero} \qquad \frac{\Gamma \vdash t_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{succ}\ t_1 : \mathtt{Nat}}\ \text{T-Succ}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{iszero}\ t_1 : \mathtt{Bool}}\ \text{T-IsZero} \qquad \frac{\Gamma \vdash t_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{pred}\ t_1 : \mathtt{Nat}}\ \text{T-Pred}$$

Similarity between Lists and Natural Numbers

Question

Do you notice that the **structures** and **rules** for lists and natural numbers are very similar?

Introduction Forms

Terms that **introduce** (or **construct**) values of a certain type.

- Boolean lists: `nil` and `cons` t t
- Natural numbers: `0` and `succ` t

Elimination Forms

Terms that **eliminate** (or **destruct**) values of a certain type.
They tell us how to **use** those values.

- Boolean lists: `isnil` t, `head` t, and `tail` t
- Natural numbers: `iszero` t and `pred` t

Unifying Introduction Forms for A Type

It would be useful to unify multiple introduction forms into a single one.

Boolean Lists

A Boolean list is either (i) an empty list `nil`, or (ii) a `cons` list of a Boolean and a Boolean list.

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} + \texttt{Bool} \times \texttt{BoolList}}{\Gamma \vdash \texttt{fold}\ [\texttt{BoolList}]\ t_1 : \texttt{BoolList}} \text{ T-Fold-BoolList}$$

We use **sum types** to unify multiple possibilities.
That is, `Unit` stands for case i and $\texttt{Bool} \times \texttt{BoolList}$ stands for case ii.

Remark (Sum Types)

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{inl}\ t_1 : T_1 + T_2} \text{ T-Inl} \qquad \frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \texttt{inr}\ t_1 : T_1 + T_2} \text{ T-Inr}$$

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \qquad \Gamma, x_1 : T_1 \vdash t_1 : T \qquad \Gamma, x_2 : T_2 \vdash t_2 : T}{\Gamma \vdash \texttt{case}\ t_0\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow t_1 \mid \texttt{inr}\ x_2 \Rightarrow t_2 : T} \text{ T-Case}$$

Unifying Introduction Forms for A Type

Natural Numbers

A natural number is either (i) zero `0`, or (ii) a `succ` of a natural number.

$$\frac{\Gamma \vdash t_1 : \mathtt{Unit} + \mathtt{Nat}}{\Gamma \vdash \mathtt{fold}\ [\mathtt{Nat}]\ t_1 : \mathtt{Nat}}\ \text{T-Fold-Nat}$$

Similarly, `Unit` stands for case i and `Nat` stands for case ii.

Example

$$\mathtt{0} \equiv \mathtt{fold}\ [\mathtt{Nat}]\ (\mathtt{inl}\ \mathtt{unit})$$
$$\mathtt{succ}\ t \equiv \mathtt{fold}\ [\mathtt{Nat}]\ (\mathtt{inr}\ t)$$

$$\mathtt{nil} \equiv \mathtt{fold}\ [\mathtt{BoolList}]\ (\mathtt{inl}\ \mathtt{unit})$$
$$\mathtt{cons}\ t_1\ t_2 \equiv \mathtt{fold}\ [\mathtt{BoolList}]\ (\mathtt{inr}\ \{t_1, t_2\})$$

Generalizing the fold Operator

Question

Can we **inline** the meaning of BoolList into fold?

Recursion Operator μ

We can think of BoolList as a type satisfying the equation $\texttt{BoolList} = \texttt{Unit} + \texttt{Bool} \times \texttt{BoolList}$.
Abstractly, it is a solution to the equation $X = \texttt{Unit} + \texttt{Bool} \times X$. Let us denote it by $\mu X.\, \texttt{Unit} + \texttt{Bool} \times X$.

Principle

Let us write $\texttt{fold}\ [X.\, \texttt{Unit} + \texttt{Bool} \times X]$ for $\texttt{fold}\ [\texttt{BoolList}]$.

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} + \texttt{Bool} \times (\mu X.\, \texttt{Unit} + \texttt{Bool} \times X)}{\Gamma \vdash \texttt{fold}\ [X.\, \texttt{Unit} + \texttt{Bool} \times X]\ t_1 : \mu X.\, \texttt{Unit} + \texttt{Bool} \times X} \text{ T-Fold-BoolList}$$

$$\frac{\Gamma \vdash t_1 : [X \mapsto \mu X.\, T]T}{\Gamma \vdash \texttt{fold}\ [X.\, T]\ t_1 : \mu X.\, T} \text{ T-Fold}$$

Generalizing the fold Operator

$$\frac{\Gamma \vdash t_1 : [X \mapsto \mu X.\, T]T}{\Gamma \vdash \mathtt{fold}\ [X.\, T]\ t_1 : \mu X.\, T} \text{ T-Fold}$$

Example (Boolean Lists)

$$\mathtt{BoolList} \equiv \mu X.\, \mathtt{Unit} + \mathtt{Bool} \times X$$

$$\mathtt{nil} \equiv \mathtt{fold}\ [X.\, \mathtt{Unit} + \mathtt{Bool} \times X]\ (\mathtt{inl}\ \mathtt{unit})$$

$$\mathtt{cons}\ t_1\ t_2 \equiv \mathtt{fold}\ [X.\, \mathtt{Unit} + \mathtt{Bool} \times X]\ (\mathtt{inr}\ \{t_1, t_2\})$$

Example (Natural Numbers)

$$\mathtt{Nat} \equiv \mu X.\, \mathtt{Unit} + X$$

$$\mathtt{0} \equiv \mathtt{fold}\ [X.\, \mathtt{Unit} + X]\ (\mathtt{inl}\ \mathtt{unit})$$

$$\mathtt{succ}\ t \equiv \mathtt{fold}\ [X.\, \mathtt{Unit} + X]\ (\mathtt{inr}\ t)$$

Recursive Types

The types we worked on so far (e.g., $\mathtt{BoolList}$ and $\mathtt{Nat}$) are **recursive** types.

Observation

Every value of a recursive type is the **folding** of a value of the **unfolding** of the recursive type.

$$\frac{\Gamma \vdash t_1 : [X \mapsto \mu X.\, T]T}{\Gamma \vdash \mathtt{fold}\ [X.\, T]\ t_1 : \mu X.\, T}\ \text{T-Fold}$$

Solving the Type Equation

Let $[\![T]\!]$ be the set of values of type T, e.g., $[\![\mathtt{Unit}]\!] = \{\mathtt{unit}\}$, $[\![\mathtt{Bool}]\!] = \{\mathtt{true}, \mathtt{false}\}$.
Consider $\mathtt{BoolList}$. The solution $[\![X]\!]$ to the equation $X = \mathtt{Unit} + \mathtt{Bool} \times X$ should satisfy:

$$[\![X]\!] \cong \{\mathtt{inl}\ \mathtt{unit}\} \cup \{\mathtt{inr}\ \{v_1, v_2\} \mid v_1 \in [\![\mathtt{Bool}]\!], v_2 \in [\![X]\!]\}$$

Principle

Recursive types denote the solutions to type equations.

Unifying Elimination Forms for A Type

Remark

Recall that elimination forms **destruct** values of a certain type.

Observation

For the type $\mu X.\, T$, the operator $\texttt{fold}\ [X.\, T]$ can be thought of as a function with type $[X \mapsto \mu X.\, T]T \to \mu X.\, T$.

- Boolean lists: $\texttt{fold}\ [X.\, \texttt{Unit} + \texttt{Bool} \times X] : \texttt{Unit} + \texttt{Bool} \times \texttt{BoolList} \to \texttt{BoolList}$
- Natural numbers: $\texttt{fold}\ [X.\, \texttt{Unit} + X] : \texttt{Unit} + \texttt{Nat} \to \texttt{Nat}$

Principle

Elimination forms are the **inverse** of introduction forms.

- Boolean lists: the elimination form has type $\texttt{BoolList} \to \texttt{Unit} + \texttt{Bool} \times \texttt{BoolList}$.
- Natural numbers: the elimination form has type $\texttt{Nat} \to \texttt{Unit} + \texttt{Nat}$

In general, the elimination forms have type $\mu X.\, T \to [X \mapsto \mu X.\, T]T$.

Unifying Elimination Forms for A Type

Principle

For the type $\mu X.\, T$, its elimination form has type $\mu X.\, T \to [X \mapsto \mu X.\, T]T$.

$$\frac{\Gamma \vdash t_1 : [X \mapsto \mu X.\, T]T}{\Gamma \vdash \texttt{fold}\ [X.\, T]\ t_1 : \mu X.\, T}\ \text{T-Fold} \qquad \frac{\Gamma \vdash t_1 : \mu X.\, T}{\Gamma \vdash \texttt{unfold}\ [X.\, T]\ t_1 : [X \mapsto \mu X.\, T]T}\ \text{T-Unfold}$$

Example (Boolean Lists)

$$\frac{\Gamma \vdash t_1 : \texttt{BoolList}}{\Gamma \vdash \texttt{unfold}\ [X.\, \texttt{Unit} + \texttt{Bool} \times X]\ t_1 : \texttt{Unit} + \texttt{Bool} \times \texttt{BoolList}}\ \text{T-Unfold-BoolList}$$

$$\begin{aligned}
\texttt{isnil}\ t &\equiv \texttt{case}\ \texttt{unfold}\ [X.\, \texttt{Unit} + \texttt{Bool} \times X]\ t\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow \texttt{true} \mid \texttt{inr}\ x_2 \Rightarrow \texttt{false} \\
\texttt{head}\ t &\equiv \texttt{case}\ \texttt{unfold}\ [X.\, \texttt{Unit} + \texttt{Bool} \times X]\ t\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow \texttt{error} \mid \texttt{inr}\ x_2 \Rightarrow x_2.1 \\
\texttt{tail}\ t &\equiv \texttt{case}\ \texttt{unfold}\ [X.\, \texttt{Unit} + \texttt{Bool} \times X]\ t\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow \texttt{error} \mid \texttt{inr}\ x_2 \Rightarrow x_2.2
\end{aligned}$$

Unifying Elimination Forms for A Type

Principle

For the type $\mu X.\, T$, its elimination form has type $\mu X.\, T \to [X \mapsto \mu X.\, T]T$.

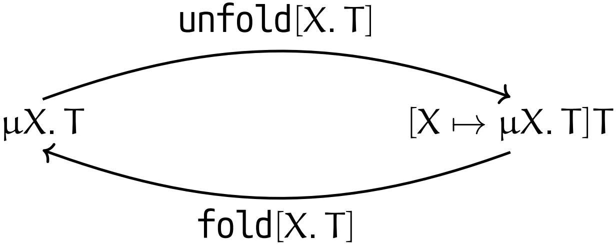
$$\frac{\Gamma \vdash t_1 : [X \mapsto \mu X.\, T]T}{\Gamma \vdash \texttt{fold}\ [X.\, T]\ t_1 : \mu X.\, T}\ \text{T-Fold} \qquad \frac{\Gamma \vdash t_1 : \mu X.\, T}{\Gamma \vdash \texttt{unfold}\ [X.\, T]\ t_1 : [X \mapsto \mu X.\, T]T}\ \text{T-Unfold}$$

Example (Natural Numbers)

$$\frac{\Gamma \vdash t_1 : \texttt{Nat}}{\Gamma \vdash \texttt{unfold}\ [X.\, \texttt{Unit} + X]\ t_1 : \texttt{Unit} + \texttt{Nat}}\ \text{T-Unfold-Nat}$$

$$\texttt{iszero}\ t \equiv \texttt{case}\ \texttt{unfold}\ [X.\, \texttt{Unit} + X]\ t\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow \texttt{true} \mid \texttt{inr}\ x_2 \Rightarrow \texttt{false}$$
$$\texttt{pred}\ t \equiv \texttt{case}\ \texttt{unfold}\ [X.\, \texttt{Unit} + X]\ t\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow 0 \mid \texttt{inr}\ x_2 \Rightarrow x_2$$

The Iso-Recursive Approach



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

Question

Use the iso-recursive approach to formulate a type for binary trees containing a Boolean in each internal node.

Question

OCaml/MoonBit uses iso-recursive types (by default). Where are the `fold`'s and `unfold`'s?

Examples of Recursive Types

Remark

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

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

Example

Let us revisit Boolean lists and natural numbers.

$$\texttt{BoolList} \equiv \mu X.\texttt{<nil}:\texttt{Unit},\texttt{cons}:\{\texttt{Bool}, X\}\texttt{>}$$

$$\texttt{Nat} \equiv \mu X.\texttt{<zero}:\texttt{Unit},\texttt{succ}:X\texttt{>}$$

Lists with Natural-Number Elements

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

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

isnil = λl:NatList. case unfold [NatList] l of <nil=u> ⇒ true | <cons=p> ⇒ false;
▶ isnil : NatList → Bool
head = λl:NatList. case unfold [NatList] l of <nil=u> ⇒ error | <cons=p> ⇒ p.1;
▶ head : NatList → Nat
tail = λl:NatList. case unfold [NatList] l of <nil=u> ⇒ error | <cons=p> ⇒ p.2;
▶ tail : NatList → NatList

sumlist = fix (λs:NatList→Nat. λl:NatList.
                  if isnil l then 0 else plus (head l) (s (tail 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. fold [Hungry] f);
▶ f : Nat→Hungry

f 0;
▶ fold [Hungry] <fun> : Hungry

unfold [Hungry] (f 0);
▶ <fun> : Nat→Hungry

unfold [Hungry] (unfold [Hungry] (f 0) 1) 2;
▶ fold [Hungry] <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};

head = λs:Stream. (unfold [Stream] s unit).1;
▶ head : Stream → Nat
tail = λs:Stream. (unfold [Stream] s unit).2;
▶ tail : Stream → Stream

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

Question

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

Streams

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

head fib;
▶ 1 : Nat
head (tail (tail (tail fib)));
▶ 3 : Nat
head (tail (tail (tail (tail (tail (tail 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};

c1 = let create = fix (λf:{x:Nat}→Counter. λs:{x:Nat}.
                          fold [Counter]
                            {get = s.x,
                             inc = λ_:Unit. f {x=succ(s.x)},
                             dec = λ_:Unit. f {x=pred(s.x)} })
     in create {x=0};
▶ c1 : Counter

c2 = (unfold [Counter] c1).inc unit;
▶ c2 : Counter
(unfold [Counter] c2).get;
▶ 1 : 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

```
Div_T = μA. A→T;
omega_T = (λx:Div_T. unfold [Div_T] x x) (fold [Div_T] (λx:Div_T. unfold [Div_T] x x));
▶ omega_T : T
```

Recursive types break the **strong-normalization** property (c.f., Chapter 12) **without** using fixed points!

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?

$$
\begin{array}{l}
\mathsf{Y}_T = \lambda f{:}T{\to}T. \\
\quad (\lambda x{:}\mathsf{Div}_T.\ f\ (\mathbf{unfold}\ [\mathsf{Div}_T]\ x\ x))\ (\mathbf{fold}\ [\mathsf{Div}_T]\ (\lambda x{:}\mathsf{Div}_T.\ f\ (\mathbf{unfold}\ [\mathsf{Div}_T]\ x\ x)));
\end{array}
$$

$\blacktriangleright\ \mathsf{Y}_T : (T{\to}T) \to T$

Question (Homework)

Implement Y_T in OCaml/MoonBit. 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 three recursive functions.

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. fold [D] f;
▶ lam : (D→D) → D

ap = λf:D. λa:D. unfold [D] f a;
▶ ap : D → D → D
```

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

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

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

Syntactic Forms

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

Typing and Evaluation Rules

$$\frac{\Gamma \vdash t_1 : [X \mapsto \mu X.\,T_1]T_1}{\Gamma \vdash \texttt{fold}\ [X.\,T_1]\ t_1 : \mu X.\,T_1}\ \text{T-Fold} \qquad \frac{\Gamma \vdash t_1 : \mu X.\,T_1}{\Gamma \vdash \texttt{unfold}\ [X.\,T_1]\ t_1 : [X \mapsto \mu X.\,T_1]T_1}\ \text{T-Unfold}$$

$$\frac{}{\texttt{unfold}\ [X.\,S]\ (\texttt{fold}\ [Y.\,T]\ v_1) \longrightarrow v_1}\ \text{E-UnfoldFold}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{fold}\ [X.\,T]\ t_1 \longrightarrow \texttt{fold}\ [X.\,T]\ t_1'}\ \text{E-Fold} \qquad \frac{t_1 \longrightarrow t_1'}{\texttt{unfold}\ [X.\,T]\ t_1 \longrightarrow \texttt{unfold}\ [X.\,T]\ t_1'}\ \text{E-Unfold}$$

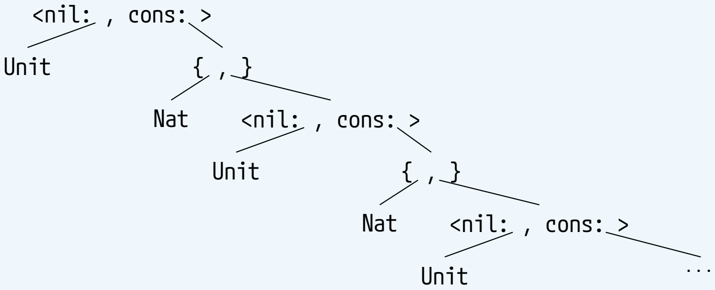
Another Approach to Recursive Types

Question

Let us revisit the question: what is the relation between the type $\mu X.\, T$ and its one-step unfolding $[X \mapsto \mu X.\, T]T$?

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

NatList as An Infinite Tree



Another Approach to Recursive Types

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

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.

The Equi-Recursive Approach

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

Lists under Equi-Recursive Types

```
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
head = λl:NatList. case l of <nil=u> ⇒ error | <cons=p> ⇒ p.1;
▶ head : NatList → Nat
tail = λl:NatList. case l of <nil=u> ⇒ error | <cons=p> ⇒ p.2;
```

Question

Re-implement previous examples of iso-recursive types under equi-recursive types.

Recursive Types are Useless as Logics

Remark (Curry-Howard Correspondence)

In simply-typed lambda-calculus, we can interpret types as logical propositions (c.f., Chapter 9).

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.

$$\texttt{omega}_T = (\lambda x{:}(\mu A.\ A{\to}T).\ x\ x)\ (\lambda x{:}(\mu A.\ A{\to}T).\ x\ x);$$
$$\blacktriangleright\ \texttt{omega}_T : T$$

The fact that $\texttt{omega}_T$ 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

Suppose that we are not allowed to use fixed points.
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	$\mu A.\, A \to \texttt{Nat}$	✗
Untyped lambda-calculus	$\mu X.\, X \to X$	✗

Observation

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

Positive Type Operators

$X. T$ pos: "type operator $X. T$ is positive"

$$\frac{}{X. X \text{ pos}} \qquad \frac{}{X. \texttt{Unit} \text{ pos}} \qquad \frac{X. T_1 \text{ pos} \qquad X. T_2 \text{ pos}}{X. T_1 \times T_2 \text{ pos}} \qquad \frac{X. T_1 \text{ pos} \qquad X. T_2 \text{ pos}}{X. T_1 + T_2 \text{ pos}}$$

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

Question

Which of the following type operators are positive?

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

Inductive & Coinductive Types

Positive type operators can be used to build **inductive** and **coinductive** types.

Syntactic Forms

$$T ::= \ldots \mid X \mid \mathtt{ind}(X.\,T) \mid \mathtt{coi}(X.\,T) \qquad \text{where } X.\,T \text{ pos}$$

$$t ::= \ldots \mid \mathtt{fold}\ [X.\,T]\ t \mid \mathtt{unfold}\ [X.\,T]\ t$$

Remark (Solving the Type Equation)

Let $[\![T]\!]$ be the set of values of type T, e.g., $[\![\mathtt{Unit}]\!] = \{\mathtt{unit}\}$, $[\![\mathtt{Bool}]\!] = \{\mathtt{true}, \mathtt{false}\}$.
Consider $\mathtt{BoolList}$. The solution $[\![X]\!]$ to the equation $X = \mathtt{Unit} + \mathtt{Bool} \times X$ should satisfy:

$$[\![X]\!] \cong \{\mathtt{inl}\ \mathtt{unit}\} \cup \{\mathtt{inr}\ \{v_1, v_2\} \mid v_1 \in [\![\mathtt{Bool}]\!], v_2 \in [\![X]\!]\}$$

Principle

Inductive types are the **least** solutions. For example, the least solution to $X = \mathtt{Unit} + X$ is isomorphic to $\mathbb{N}$.
Coinductive types are the **greatest** solutions.

Well-Founded Recursion for Inductive Types

Question

How to compute the length of a Boolean list?
Can you do that **without** using fixed points?

Question

Is there a way to allow **useful** recursion schemes on Boolean lists, without allowing general fixed points?

Principle (Structural Recursion)

The argument of a recursion function call can only be the **sub-structures** of the function parameter.

$$\mathtt{len}\ \mathtt{t} = \mathtt{case}\ \mathtt{unfold}\ [X.\ \mathtt{Unit} + \mathtt{Bool} \times X]\ \mathtt{t}\ \mathtt{of}\ \mathtt{inl}\ x_1 \Rightarrow \mathtt{0} \mid \mathtt{inr}\ x_2 \Rightarrow \mathtt{succ}\ (\mathtt{len}\ x_2.2)$$

It is just **iteration**!

An Iteration Operator for Boolean Lists

Remark (Specialized Introduction Form)

$$\frac{\Gamma \vdash t_1 : \mathtt{Unit} + \mathtt{Bool} \times \mathtt{BoolList}}{\Gamma \vdash \mathtt{fold}\ [\mathtt{BoolList}]\ t_1 : \mathtt{BoolList}} \text{ T-Fold-BoolList}$$

Principle (Structural Recursion via Iteration)

$$\frac{\Gamma \vdash t_1 : \mathtt{BoolList} \qquad \Gamma, x : \mathtt{Unit} + \mathtt{Bool} \times S \vdash t_2 : S}{\Gamma \vdash \mathbf{iter}\ [\mathtt{BoolList}]\ t_1\ \mathbf{with}\ x.\, t_2 : S} \text{ T-Iter-BoolList}$$

$$\frac{}{\mathbf{iter}\ [\mathtt{BoolList}]\ (\mathtt{fold}\ [\mathtt{BoolList}]\ v)\ \mathbf{with}\ x.\, t_2 \longrightarrow t'} \text{ E-Iter-BoolList}$$

where

$$\begin{aligned} t' \equiv \mathbf{let}\ x = \mathtt{case}\ v\ \mathtt{of}\ & \mathtt{inl}\ x_1 \Rightarrow \mathtt{inl}\ x_1 \mid \\ & \mathtt{inr}\ x_2 \Rightarrow \mathtt{inr}\ \{x_2.1, \mathbf{iter}\ [\mathtt{BoolList}]\ x_2.2\ \mathbf{with}\ x.\, t_2\} \\ \mathbf{in}\ t_2 & \end{aligned}$$

An Iteration Operator for Boolean Lists

$$\frac{\Gamma \vdash t_1 : \mathtt{BoolList} \qquad \Gamma, x : \mathtt{Unit} + \mathtt{Bool} \times S \vdash t_2 : S}{\Gamma \vdash \textbf{iter}\ [\mathtt{BoolList}]\ t_1\ \textbf{with}\ x.\, t_2 : S} \text{ T-Iter-BoolList}$$

Example

$$\mathtt{isnil}\ t \equiv \textbf{iter}\ [\mathtt{BoolList}]\ t\ \textbf{with}\ x.\ \mathtt{case}\ x\ \mathtt{of}\ \mathtt{inl}\ x_1 \Rightarrow \mathtt{true} \mid \mathtt{inr}\ x_2 \Rightarrow \mathtt{false}$$

$$\mathtt{len}\ t \equiv \textbf{iter}\ [\mathtt{BoolList}]\ t\ \textbf{with}\ x.\ \mathtt{case}\ x\ \mathtt{of}\ \mathtt{inl}\ x_1 \Rightarrow 0 \mid \mathtt{inr}\ x_2 \Rightarrow \mathtt{succ}\ x_2.2$$

Question

Write down the evaluation of $\mathtt{len}\ \ell_2$ where:

$$\ell_2 \equiv \mathtt{fold}\ [\mathtt{BoolList}]\ (\mathtt{inr}\ \{\mathtt{true}, \ell_1\})$$
$$\ell_1 \equiv \mathtt{fold}\ [\mathtt{BoolList}]\ (\mathtt{inr}\ \{\mathtt{false}, \ell_0\})$$
$$\ell_0 \equiv \mathtt{fold}\ [\mathtt{BoolList}]\ (\mathtt{inl}\ \mathtt{unit})$$

An Iteration Operator for Natural Numbers

Let us repeat the same development for the inductive type of natural numbers.

$$\frac{\Gamma \vdash t_1 : \mathsf{Unit} + \mathsf{Nat}}{\Gamma \vdash \mathtt{fold}\ [\mathsf{Nat}]\ t_1 : \mathsf{Nat}} \ \text{T-Fold-Nat}$$

Now consider **iteration** over natural numbers.

$$\frac{\Gamma \vdash t_1 : \mathsf{Nat} \qquad \Gamma, x : \mathsf{Unit} + S \vdash t_2 : S}{\Gamma \vdash \mathbf{iter}\ [\mathsf{Nat}]\ t_1\ \mathbf{with}\ x.\, t_2 : S} \ \text{T-Iter-Nat}$$

$$\frac{}{\mathbf{iter}\ [\mathsf{Nat}]\ (\mathtt{fold}\ [\mathsf{Nat}]\ v)\ \mathbf{with}\ x.\, t_2 \longrightarrow t'} \ \text{E-Iter-Nat}$$

where

$$\begin{aligned} t' \equiv \mathbf{let}\ x = \mathtt{case}\ v\ \mathtt{of}\ & \mathtt{inl}\ x_1 \Rightarrow \mathtt{inl}\ x_1 \mid \\ & \mathtt{inr}\ x_2 \Rightarrow \mathtt{inr}\ (\mathbf{iter}\ [\mathsf{Nat}]\ x_2\ \mathbf{with}\ x.\, t_2) \\ \mathbf{in}\ t_2 & \end{aligned}$$

Generalizing the iter Operator

Question

Can we **inline** the meaning of BoolList into **iter**?

Principle

Let us write **iter** $[X.\,\mathtt{Unit} + \mathtt{Bool} \times X]$ for **iter** $[\mathtt{BoolList}]$.

$$\frac{\Gamma \vdash t_1 : \mathtt{ind}(X.\,\mathtt{Unit} + \mathtt{Bool} \times X) \qquad \Gamma, x : \mathtt{Unit} + \mathtt{Bool} \times S \vdash t_2 : S}{\Gamma \vdash \mathbf{iter}\ [X.\,\mathtt{Unit} + \mathtt{Bool} \times X]\ t_1\ \mathbf{with}\ x.\,t_2 : S} \text{ T-Iter-BoolList}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{ind}(X.\,T) \qquad \Gamma, x : [X \mapsto S]T \vdash t_2 : S}{\Gamma \vdash \mathbf{iter}\ [X.\,T]\ t_1\ \mathbf{with}\ x.\,t_2 : S} \text{ T-Iter}$$

Generalizing the iter Operator

$$\frac{\Gamma \vdash t_1 : \mathtt{ind}(X.\,T) \qquad \Gamma, x : [X \mapsto S]T \vdash t_2 : S}{\Gamma \vdash \mathbf{iter}\ [X.\,T]\ t_1\ \mathbf{with}\ x.\,t_2 : S}\ \text{T-Iter}$$

Principle

Let us write $\mathtt{fold}\ [X.\,\mathtt{Unit} + \mathtt{Bool} \times X]$ for $\mathtt{fold}\ [\mathtt{BoolList}]$.

$$\frac{\Gamma \vdash t_1 : \mathtt{Unit} + \mathtt{Bool} \times \mathtt{ind}(X.\,\mathtt{Unit} + \mathtt{Bool} \times X)}{\Gamma \vdash \mathtt{fold}\ [X.\,\mathtt{Unit} + \mathtt{Bool} \times X]\ t_1 : \mathtt{ind}(X.\,\mathtt{Unit} + \mathtt{Bool} \times X)}\ \text{T-Fold-BoolList}$$

$$\frac{\Gamma \vdash t_1 : [X \mapsto \mathtt{ind}(X.\,T)]T}{\Gamma \vdash \mathtt{fold}\ [X.\,T]\ t_1 : \mathtt{ind}(X.\,T)}\ \text{T-Fold}$$

Question

What about the evaluation rules for **iter**?

Generalizing the iter Operator

$$\frac{\Gamma \vdash t_1 : \mathrm{ind}(X.\, T) \qquad \Gamma, x : [X \mapsto S]T \vdash t_2 : S}{\Gamma \vdash \textbf{iter}\ [X.\, T]\ t_1\ \textbf{with}\ x.\, t_2 : S} \text{ T-Iter}$$

$$\frac{}{\textbf{iter}\ [X.\, \mathrm{Unit} + \mathrm{Bool} \times X]\ (\mathrm{fold}\ [X.\, \mathrm{Unit} + \mathrm{Bool} \times X]\ v)\ \textbf{with}\ x.\, t_2 \longrightarrow t'} \text{ E-Iter-BoolList}$$

where

$$\begin{aligned} t' \equiv \textbf{let}\ x = \mathrm{case}\ v\ \mathrm{of}\ & \mathrm{inl}\ x_1 \Rightarrow \mathrm{inl}\ x_1 \mid \\ & \mathrm{inr}\ x_2 \Rightarrow \mathrm{inr}\ \{x_2.1, \textbf{iter}\ [X.\, \mathrm{Unit} + \mathrm{Bool} \times X]\ x_2.2\ \textbf{with}\ x.\, t_2\} \\ \textbf{in}\ t_2 & \end{aligned}$$

Observation

$\textbf{iter}\ [X.\, T]\ (\mathrm{fold}\ [X.\, T]\ v)\ \textbf{with}\ x.\, t_2$ should replace every sub-structure v_{sub} of v that corresponds to an occurrence of X in T by $\textbf{iter}\ [X.\, T]\ v_{sub}\ \textbf{with}\ x.\, t_2$.

Generalizing the iter Operator

Observation

$\texttt{iter}\ [X.\,T]\ (\texttt{fold}\ [X.\,T]\ v)\ \texttt{with}\ x.\,t_2$ should replace every sub-structure v_{sub} of v that corresponds to an occurrence of X in T by $\texttt{iter}\ [X.\,T]\ v_{\text{sub}}\ \texttt{with}\ x.\,t_2$.

Principle

$$\frac{}{\texttt{iter}\ [X.\,T]\ (\texttt{fold}\ [X.\,T]\ v)\ \texttt{with}\ x.\,t_2 \longrightarrow [x \mapsto \texttt{map}\ [X.\,T]\ v\ \texttt{with}\ y.\,(\texttt{iter}\ [X.\,T]\ y\ \texttt{with}\ x.\,t_2)]t_2}\ \text{E-Iter}$$

The operator **map** is defined **inductively** on the structure of the **positive** type operator.

$$\frac{}{\texttt{map}\ [X.\,X]\ v\ \texttt{with}\ y.\,t_2 \longrightarrow [y \mapsto v]t_2}\ \text{E-Map-Var}$$

$$\frac{}{\texttt{map}\ [X.\,\texttt{Unit}]\ v\ \texttt{with}\ y.\,t_2 \longrightarrow v}\ \text{E-Map-Unit}$$

$$\frac{}{\texttt{map}\ [X.\,T_1 \times T_2]\ v\ \texttt{with}\ y.\,t_2 \longrightarrow \{\texttt{map}\ [X.\,T_1]\ v.1\ \texttt{with}\ y.\,t_2, \texttt{map}\ [X.\,T_2]\ v.2\ \texttt{with}\ y.\,t_2\}}\ \text{E-Map-Prod}$$

Generalizing the `iter` Operator

Principle (Generic Mapping)

$$\frac{}{\textbf{map}\ [X.\, X]\ v\ \textbf{with}\ y.\, t_2 \longrightarrow [y \mapsto v]t_2}\ \text{E-Map-Var} \qquad \frac{}{\textbf{map}\ [X.\, \mathtt{Unit}]\ v\ \textbf{with}\ y.\, t_2 \longrightarrow v}\ \text{E-Map-Unit}$$

$$\frac{}{\textbf{map}\ [X.\, T_1 \times T_2]\ v\ \textbf{with}\ y.\, t_2 \longrightarrow \{\textbf{map}\ [X.\, T_1]\ v.1\ \textbf{with}\ y.\, t_2, \textbf{map}\ [X.\, T_2]\ v.2\ \textbf{with}\ y.\, t_2\}}\ \text{E-Map-Prod}$$

$$\frac{}{\textbf{map}\ [X.\, T_1 + T_2]\ v\ \textbf{with}\ y.\, t_2 \longrightarrow t'}\ \text{E-Map-Sum}$$

where

$$\begin{aligned} t' \equiv \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}\ x_1 &\Rightarrow \mathtt{inl}\ (\textbf{map}\ [X.\, T_1]\ x_1\ \textbf{with}\ y.\, t_2)\ \mid \\ \mathtt{inr}\ x_2 &\Rightarrow \mathtt{inr}\ (\textbf{map}\ [X.\, T_2]\ x_2\ \textbf{with}\ y.\, t_2) \end{aligned}$$

Question

Derive the evaluation rules `E-Iter-BoolList` and `E-Iter-Nat` from these more general rules.

Examples of Iteration for Inductive Types

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

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

Revisiting 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};

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

Observation

A stream is isomorphic to an **infinite** list.
Consider the solution $[\![X]\!]$ to the equation $X = \mathsf{Nat} \times X$. It should satisfy:

$$[\![X]\!] \cong \{\mathtt{inr}\ \{v_1, v_2\} \mid v_1 \in [\![\mathsf{Nat}]\!], v_2 \in [\![X]\!]\}$$

The **least** solution is just the empty set.
But the **greatest** solution is $\{\mathtt{inr}\ \{v_1, \mathtt{inr}\ \{v_2, \mathtt{inr}\ \{v_3, \ldots\}\}\} \mid v_1, v_2, v_3, \ldots \in [\![\mathsf{Nat}]\!]\}$, i.e., the streams!

Well-Founded Recursion for Coinductive Types

Question

What is the difference between the recursion schemes on inductive and coinductive types?

Observation

- For inductive types (e.g., lists), we use recursion to **iterate** over them.
- For coinductive types (e.g., streams), we use recursion to **generate** them.

Question

Recall the implementation of streams under general recursive types:

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

Can we define a recursion scheme for **generating** values of coinductive types?

A Generation Operator for Streams

Remark (Specialized Elimination Form)

Let us consider the type of streams as the greatest solution to $X = \text{Nat} \times X$.

$$\frac{\Gamma \vdash t_1 : \texttt{Stream}}{\Gamma \vdash \texttt{unfold}\ [\texttt{Stream}]\ t_1 : \text{Nat} \times \texttt{Stream}}\ \text{T-Unfold-Stream}$$

Principle (Structural Recursion for Generation)

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

$$\frac{}{\begin{array}{c}\texttt{unfold}\ [\texttt{Stream}]\ (\textbf{gen}\ [\texttt{Stream}]\ v\ \textbf{with}\ x.t_2) \\ \longrightarrow \\ \textbf{let}\ v_2 = [x \mapsto v]t_2\ \textbf{in}\ \{v_2.1, (\textbf{gen}\ [\texttt{Stream}]\ v_2.2\ \textbf{with}\ x.t_2)\}\end{array}}\ \text{E-Gen-Stream}$$

A Generation Operator for Streams

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

Example

$$\mathsf{upfrom0} \equiv \mathbf{gen}\ [\mathsf{Stream}]\ 0\ \mathbf{with}\ x.\ \{x, \mathsf{succ}\ x\}$$
$$\mathsf{fib} \equiv \mathbf{gen}\ [\mathsf{Stream}]\ \{1, 1\}\ \mathbf{with}\ x.\ \{x.1, \{x.2, (\mathsf{plus}\ x.1\ x.2)\}\}$$

Question

Write down the evaluation of $(\mathsf{unfold}\ [\mathsf{Stream}]\ t_2).1$ where:

$$t_2 \equiv (\mathsf{unfold}\ [\mathsf{Stream}]\ t_1).2$$
$$t_1 \equiv (\mathsf{unfold}\ [\mathsf{Stream}]\ t_0).2$$
$$t_0 \equiv (\mathsf{unfold}\ [\mathsf{Stream}]\ \mathsf{fib}).2$$

Generalizing the gen Operator

Question

Can we **inline** the meaning of `Stream` (i.e., the greatest solution to $X = \mathsf{Nat} \times X$) into **gen**?

Principle

Let us write **gen** $[X.\, \mathsf{Nat} \times X]$ for **gen** $[\mathtt{Stream}]$.

$$\frac{\Gamma \vdash t_1 : S \qquad \Gamma, x : S \vdash t_2 : \mathsf{Nat} \times S}{\Gamma \vdash \mathbf{gen}\ [X.\, \mathsf{Nat} \times X]\ t_1\ \mathbf{with}\ x.\, t_2 : \mathtt{coi}(X.\, \mathsf{Nat} \times X)} \text{ T-Gen-Stream}$$

$$\frac{\Gamma \vdash t_1 : S \qquad \Gamma, x : S \vdash t_2 : [X \mapsto S]T}{\Gamma \vdash \mathbf{gen}\ [X.\, T]\ t_1\ \mathbf{with}\ x.\, t_2 : \mathtt{coi}(X.\, T)} \text{ T-Gen}$$

Generalizing the gen Operator

$$\frac{\Gamma \vdash t_1 : S \qquad \Gamma, x : S \vdash t_2 : [X \mapsto S]T}{\Gamma \vdash \textbf{gen}\ [X.\,T]\ t_1\ \textbf{with}\ x.\,t_2 : \texttt{coi}(X.\,T)} \text{ T-Gen}$$

Principle

Let us write $\texttt{unfold}\ [X.\,\text{Nat} \times X]$ for $\texttt{unfold}\ [\texttt{Stream}]$.

$$\frac{\Gamma \vdash t_1 : \texttt{coi}(X.\,\text{Nat} \times X)}{\Gamma \vdash \texttt{unfold}\ [X.\,\text{Nat} \times X]\ t_1 : \text{Nat} \times \texttt{coi}(X.\,\text{Nat} \times X)} \text{ T-Unfold-Stream}$$

$$\frac{\Gamma \vdash t_1 : \texttt{coi}(X.\,T)}{\Gamma \vdash \texttt{unfold}\ [X.\,T]\ t_1 : [X \mapsto \texttt{coi}(X.\,T)]T} \text{ T-Unfold}$$

Question

What about the evaluation rules for unfolding a **gen**?

Generalizing the gen Operator

$$\frac{\Gamma \vdash t_1 : S \qquad \Gamma, x : S \vdash t_2 : [X \mapsto S]T}{\Gamma \vdash \textbf{gen}\ [X.\,T]\ t_1\ \textbf{with}\ x.\,t_2 : \texttt{coi}(X.\,T)} \quad \text{T-Gen}$$

$$\frac{}{\texttt{unfold}\ [X.\,\mathsf{Nat} \times X]\ (\textbf{gen}\ [X.\,\mathsf{Nat} \times X]\ v\ \textbf{with}\ x.t_2) \longrightarrow \textbf{let}\ v_2 = [x \mapsto v]t_2\ \textbf{in}\ \{v_2.1, (\textbf{gen}\ [X.\,\mathsf{Nat} \times X]\ v_2.2\ \textbf{with}\ x.t_2)\}} \quad \text{E-Gen-Stream}$$

Observation

$\texttt{unfold}\ [X.\,T]\ (\textbf{gen}\ [X.\,T]\ v\ \textbf{with}\ x.\,t_2)$ should substitute x with v in t_2, obtain the result v_2, and replace every sub-structure v_{sub} of v_2 that corresponds to an occurrence of X in T by $\textbf{gen}\ [X.\,T]\ v_{sub}\ \textbf{with}\ x.\,t_2$.

Generalizing the gen Operator

Observation

$\texttt{unfold}\ [X.\,T]\ (\textbf{gen}\ [X.\,T]\ v\ \textbf{with}\ x.\,t_2)$ should substitute x with v in t_2, obtain the result v_2, and replace every sub-structure v_{sub} of v_2 that corresponds to an occurrence of X in T by $\textbf{gen}\ [X.\,T]\ v_{\text{sub}}\ \textbf{with}\ x.\,t_2$.

Principle

Recall that for any **positive** type operator $X.\,T$, the term $\textbf{map}\ [X.\,T]\ v\ \textbf{with}\ y.\,t$ replaces every sub-structure v_{sub} of v that corresponds to an occurrence of X in T by $[y \mapsto v_{\text{sub}}]t$.

$$\frac{}{\texttt{unfold}\ [X.\,T]\ (\textbf{gen}\ [X.\,T]\ v\ \textbf{with}\ x.\,t_2) \longrightarrow \textbf{map}\ [X.\,T]\ [x \mapsto v]t_2\ \textbf{with}\ y.\,(\textbf{gen}\ [X.\,T]\ y\ \textbf{with}\ x.\,t_2)}\ \text{E-Gen}$$

Question

Derive the evaluation rule `E-Gen-Stream` from this more general rule.

Formulation of Inductive/Coinductive Types

Syntactic Forms

$$t ::= \dots \mid \texttt{fold}\ [X.\,T]\ t \mid \textbf{iter}\ [X.\,T]\ t\ \textbf{with}\ x.\,t \mid \texttt{unfold}\ [X.\,T]\ t \mid \textbf{gen}\ [X.\,T]\ t\ \textbf{with}\ x.\,t_2$$

$$v ::= \dots \mid \texttt{fold}\ [X.\,T]\ v \mid \textbf{gen}\ [X.\,T]\ v\ \textbf{with}\ x.\,t$$

$$T ::= \dots \mid X \mid \texttt{ind}(X.\,T) \mid \texttt{coi}(X.\,T) \qquad \text{where } X.\,T \text{ pos}$$

Remark

Inductive types are characterized by how to **construct** them (i.e., `fold`).
Coinductive types are characterized by how to **destruct** them (i.e., `unfold`).

Aside

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.

Revisiting General Recursive Types

Solving the Type Equation

Let $[\![\mathsf{T}]\!]$ be the set of values of type T, e.g., $[\![\texttt{Unit}]\!] = \{\texttt{unit}\}$, $[\![\texttt{Bool}]\!] = \{\texttt{true}, \texttt{false}\}$.
Consider `BoolList`. The solution $[\![X]\!]$ to the equation $X = \texttt{Unit} + \texttt{Bool} \times X$ should satisfy:

$$[\![X]\!] \cong \{\texttt{inl unit}\} \cup \{\texttt{inr } \{v_1, v_2\} \mid v_1 \in [\![\texttt{Bool}]\!], v_2 \in [\![X]\!]\}$$

Question

Does the definition mean **least** or **greatest** solution?

Principle (Types are NOT Sets)

For example, arrow types characterize **computable** functions, not **arbitrary** functions.
Otherwise, the equation $X = X \rightarrow X$ (with the understanding of **partial** functions) does not have a solution.
Formal (and unique) characterization of recursive types requires **domain theory**: S. Abramsky and A. Jung. 1995. Domain Theory. In *Handbook of Logic in Computer Science (Vol. 3): Semantic Structures*. Oxford University Press, Inc. https://dl.acm.org/doi/10.5555/218742.218744.

Revisiting General Recursive Types

Eager Semantics

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

$$\frac{}{\texttt{unfold}\ [X.\,S]\ (\texttt{fold}\ [Y.\,T]\ v_1) \longrightarrow v_1}\ \text{E-UnfoldFold} \qquad \frac{t_1 \longrightarrow t_1'}{\texttt{fold}\ [X.\,T]\ t_1 \longrightarrow \texttt{fold}\ [X.\,T]\ t_1'}\ \text{E-Fold}$$

Recursive types have an **inductive** flavor under eager semantics.
Coinductive analogues are accessible as well by using function types.

Lazy Semantics

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

$$\frac{}{\texttt{unfold}\ [X.\,S]\ (\texttt{fold}\ [Y.\,T]\ t_1) \longrightarrow t_1}\ \text{E-UnfoldFold}$$

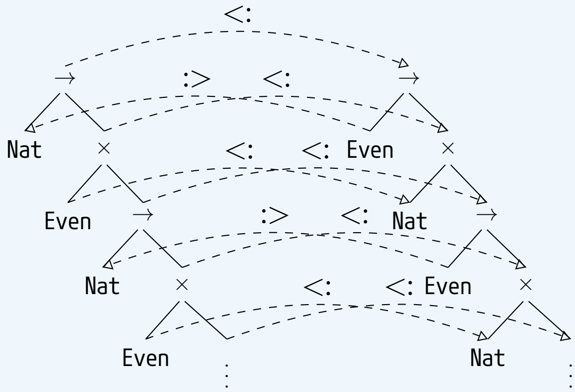
no E-Fold

Recursive types have a **coinductive** flavor under lazy semantics.
However, the inductive analogues are inaccessible.

Subtyping

Can we deduce the relation below, given that Even $<:$ Nat?

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



Review: Subtyping in Chapters 15 & 16

For brevity, we only consider three type constructors: $\to$, $\times$, and Top.

$$T ::= \text{Top} \mid T \to T \mid T \times T$$

Declarative Version

$$\frac{}{T <: \text{Top}} \qquad \frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 \times S_2 <: T_1 \times T_2}$$

$$\frac{}{S <: S} \qquad \frac{S <: U \qquad U <: T}{S <: T}$$

Algorithmic Version

$$\frac{}{\triangleright T <: \text{Top}} \qquad \frac{\triangleright T_1 <: S_1 \qquad \triangleright S_2 <: T_2}{\triangleright S_1 \to S_2 <: T_1 \to T_2} \qquad \frac{\triangleright S_1 <: T_1 \qquad \triangleright S_2 <: T_2}{\triangleright S_1 \times S_2 <: T_1 \times T_2}$$

μ-Types

Definition

Let X range over a fixed countable set $\{X_1, X_2, \ldots\}$ of type variables. The set of **raw μ-types** is the set of expressions defined by the following grammar (inductively):

$$T ::= X \mid \text{Top} \mid T \to T \mid T \times T \mid \mu X.\, T$$

Definition

A raw μ-type T is **contractive** (and called a **μ-type**) if, for any subexpression of T of the form $\mu X_1.\, \mu X_2.\, \ldots \mu X_n.\, S$, the body S is not X.

Question

How to extend the subtype relation to support μ-types?

Subtyping on μ-Types

An Attempt: μ-Folding Rules

$$\frac{S <: [X \mapsto \mu X.\, T]T}{S <: \mu X.\, T} \qquad \frac{[X \mapsto \mu X.\, S]S <: T}{\mu X.\, S <: T}$$

Question

Do those rules work?
Try those rules to check if $\mu X.\, \mathsf{Top} \times X <: \mu X.\, \mathsf{Top} \times (\mathsf{Top} \times X)$ holds.

Subtyping on μ-Types

Example

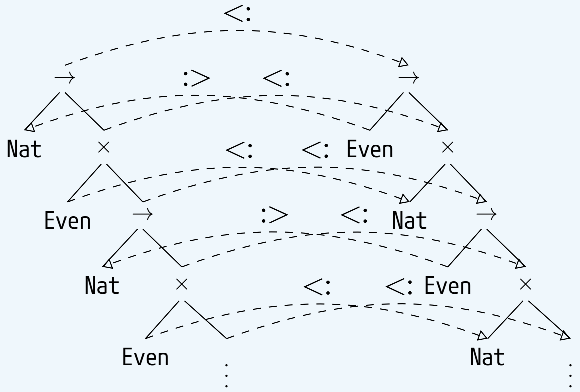
Let $S \equiv \mu X.\, \mathsf{Top} \times X$ and $T \equiv \mu X.\, \mathsf{Top} \times (\mathsf{Top} \times X)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Top} <: \mathsf{Top}} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Top} <: \mathsf{Top}} \qquad \dfrac{\vdots}{\color{red}{S <: T}}}{\mathsf{Top} \times S <: \mathsf{Top} \times T}}{S <: \mathsf{Top} \times T}}{\mathsf{Top} \times S <: \mathsf{Top} \times (\mathsf{Top} \times T)}}{\mathsf{Top} \times S <: T}}{S <: T}$$

Observation

The inference works only if we consider the subtyping rules **coinductively**, i.e., consider the **largest** relation generating by the subtyping rules.

Why?



Principle

The subtype relation must consider types with structures like **infinite trees**.

Hypothetical Subtyping

$\Sigma \vdash S <: T$: "one can derive $S <: T$ by assuming the subtype facts in Σ"

$$\frac{(S <: T) \in \Sigma}{\Sigma \vdash S <: T} \qquad \frac{}{\Sigma \vdash T <: \mathsf{Top}} \qquad \frac{\Sigma \vdash T_1 <: S_1 \qquad \Sigma \vdash S_2 <: T_2}{\Sigma \vdash S_1 \to S_2 <: T_1 \to T_2} \qquad \frac{\Sigma \vdash S_1 <: T_1 \qquad \Sigma \vdash S_2 <: T_2}{\Sigma \vdash S_1 \times S_2 <: T_1 \times T_2}$$

$$\frac{\Sigma, S <: \mu X.\, T \vdash S <: [X \mapsto \mu X.\, T]T}{\Sigma \vdash S <: \mu X.\, T} \qquad \frac{\Sigma, \mu X.\, S <: T \vdash [X \mapsto \mu X.\, S]S <: T}{\Sigma \vdash \mu X.\, S <: T}$$

Let $S \equiv \mu X.\, \mathsf{Top} \times X$ and $T \equiv \mu X.\, \mathsf{Top} \times (\mathsf{Top} \times X)$.

$$\cfrac{\cfrac{\cfrac{}{\ldots \vdash \mathsf{Top} <: \mathsf{Top}} \qquad \cfrac{\cfrac{\cfrac{}{\ldots \vdash \mathsf{Top} <: \mathsf{Top}} \qquad \cfrac{(S <: T) \in S <: T, \ldots}{S <: T, \ldots \vdash S <: T}}{S <: T, \ldots \vdash \mathsf{Top} \times S <: \mathsf{Top} \times T}}{S <: T, \ldots \vdash S <: \mathsf{Top} \times T}}{S <: T, \ldots \vdash \mathsf{Top} \times S <: \mathsf{Top} \times (\mathsf{Top} \times T)}}{\cfrac{S <: T \vdash \mathsf{Top} \times S <: T}{\varnothing \vdash S <: T}}$$

Why Does Hypothetical Subtyping Work?

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Top} <: \mathsf{Top}} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Top} <: \mathsf{Top}} \qquad \dfrac{\vdots}{\color{red}{S <: T}}}{\mathsf{Top} \times S <: \mathsf{Top} \times T}}{S <: \mathsf{Top} \times T}}{\mathsf{Top} \times S <: \mathsf{Top} \times (\mathsf{Top} \times T)}}{\mathsf{Top} \times S <: T}}{S <: T}
$$

Observation (Termination)

To check the original subtype relation $S <: T$ between μ-types, the set of reachable states $S' <: T'$ is **finite**.
See Chapter 21.9 for a detailed argument.

Question (Correctness)

Why is hypothetical subtyping correct with respect to the original (coinductive) subtype relation?

Coinductive Subtyping

Definition (Generating Functions)

A generating function is a function $F : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ that is **monotone**, i.e., $X \subseteq Y$ implies $F(X) \subseteq F(Y)$.
Let F be monotone. A subset X of $\mathcal{U}$ is a **fixed point** of F if $F(X) = X$.
The **least** fixed point is written μF. The **greatest** fixed point is written νF.[1]

Subtype Relation

Let $\mathcal{T}_m$ denote the set of all μ-types. Two μ-types S and T are said to be in the **subtype relation** ("S is a subtype of T") if $(S, T) \in \nu F_d$, where the monotone function $F_d : \mathcal{P}(\mathcal{T}_m \times \mathcal{T}_m) \to \mathcal{P}(\mathcal{T}_m \times \mathcal{T}_m)$ is defined as follows:

$$
\begin{aligned}
F_d(R) \equiv\ & \{(T, \mathsf{Top}) \mid T \in \mathcal{T}_m\} \\
& \cup \{(S_1 \to S_2, T_1 \to T_2) \mid (T_1, S_1), (S_2, T_2) \in R\} \\
& \cup \{(S_1 \times S_2, T_1 \times T_2) \mid (S_1, T_1), (S_2, T_2) \in R\} \\
& \cup \{(S, \mu X.\, T) \mid (S, [X \mapsto \mu X.\, T]T) \in R\} \cup \{(\mu X.\, S, T) \mid ([X \mapsto \mu X.\, S]S, T) \in R\}
\end{aligned}
$$

[1] Their existence and uniqueness can be justified by the Knaster-Tarski Theorem.

Correctness of Hypothetical Subtyping

Lemma

Suppose $\Sigma \vdash S <: T$ and each $S' <: T'$ in Σ satisfies $(S', T') \in \nu F_d$.
Then $(S, T) \in \nu F_d$.

Proof Sketch

By induction on the derivation of $\Sigma \vdash S <: T$.
For μ-folding rules, we need the fact that νF_d is the greatest fixed point of F_d.

Lemma

Suppose $(S, T) \in \nu F_d$. Then $\varnothing \vdash S <: T$.

Proposition

Suppose $\Sigma \vdash S <: T$ does **NOT** hold and each $S' <: T'$ in Σ satisfies $(S', T') \in \nu F_d$.
Then $(S, T) \notin \nu F_d$.

Algorithmic Hypothetical Subtyping

$\Sigma \vdash S <: T \rhd \top/\bot$: "one can/cannot derive $S <: T$ by assuming the subtype facts in Σ"

$$\frac{(S <: T) \in \Sigma}{\Sigma \vdash S <: T \rhd \top} \qquad \frac{(T, \mathsf{Top}) \notin \Sigma}{\Sigma \vdash T <: \mathsf{Top} \rhd \top} \qquad \frac{(S_1 \to S_2, T_1 \to T_2) \notin \Sigma \quad \Sigma \vdash T_1 <: S_1 \rhd \top \quad \Sigma \vdash S_2 <: T_2 \rhd \top}{\Sigma \vdash S_1 \to S_2 <: T_1 \to T_2 \rhd \top}$$

$$\frac{(S_1 \to S_2, T_1 \to T_2) \notin \Sigma \quad \Sigma \vdash T_1 <: S_1 \rhd \bot}{\Sigma \vdash S_1 \to S_2 <: T_1 \to T_2 \rhd \bot} \qquad \frac{(S_1 \to S_2, T_1 \to T_2) \notin \Sigma \quad \Sigma \vdash T_1 <: S_1 \rhd \top \quad \Sigma \vdash S_2 <: T_2 \rhd \bot}{\Sigma \vdash S_1 \to S_2 <: T_1 \to T_2 \rhd \bot}$$

$$\frac{(S, \mu X.\, T) \notin \Sigma \quad \Sigma, S <: \mu X.\, T \vdash S <: [X \mapsto \mu X.\, T]T \rhd \mathsf{ans}}{\Sigma \vdash S <: \mu X.\, T \rhd \mathsf{ans}} \qquad \frac{(\mu X.\, S, T) \notin \Sigma \quad T \neq \mathsf{Top} \quad T \neq \mu Y.\, U \quad \Sigma, \mu X.\, S <: T \vdash [X \mapsto \mu X.\, S]S <: T \rhd \mathsf{ans}}{\Sigma \vdash \mu X.\, S <: T \rhd \mathsf{ans}}$$

Otherwise, we have $\Sigma \vdash S <: T \rhd \bot$.

Proposition

Suppose $\Sigma \vdash S <: T \rhd \bot$ and each $S' <: T'$ in Σ satisfies $(S', T') \in \nu F_d$. Then $(S, T) \notin \nu F_d$.

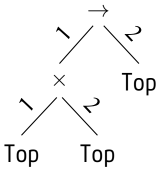
Aside: Why is Coinductive Subtyping Indeed Correct?

Definition

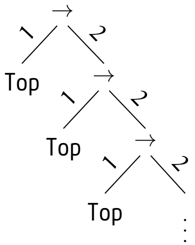
A **tree type** is a partial function $T : \{1, 2\}^* \rightharpoonup \{\rightarrow, \times, \mathsf{Top}\}$ satisfying the following constraints:

- $T(\bullet)$ is defined;
- if $T(\pi, \sigma)$ is defined then $T(\pi)$ is defined;
- if $T(\pi) = \rightarrow$ or $T(\pi) = \times$ then $T(\pi, 1)$ and $T(\pi, 2)$ are defined;
- if $T(\pi) = \mathsf{Top}$ then $T(\pi, 1)$ and $T(\pi, 2)$ are undefined.

$(\mathsf{Top} \times \mathsf{Top}) \to \mathsf{Top}$



$\mathsf{Top} \to (\mathsf{Top} \to (\mathsf{Top} \to \ldots))$



Aside: Why is Coinductive Subtyping Indeed Correct?

Subtype Relation

Let $\mathcal{T}$ denote the set of all tree types. Two tree types S and T are said to be in the **subtype relation** ("S is a subtype of T") if $(S, T) \in \nu F$, where the monotone function $F : \mathcal{P}(\mathcal{T} \times \mathcal{T}) \to \mathcal{P}(\mathcal{T} \times \mathcal{T})$ is defined as follows:

$$
\begin{aligned}
F(R) \equiv\ & \{(T, \mathsf{Top}) \mid T \in \mathcal{T}\} \\
& \cup \{(S_1 \to S_2, T_1 \to T_2) \mid (T_1, S_1), (S_2, T_2) \in R\} \\
& \cup \{(S_1 \times S_2, T_1 \times T_2) \mid (S_1, T_1), (S_2, T_2) \in R\}
\end{aligned}
$$

Principle

Under an equi-recursive setting, the subtype relation νF on possibly-infinite tree types is the desired relation.

Interpreting μ-Types as Possibly-Infinite Tree Types

The function *treeof*, mapping closed μ-types to tree types, is defined inductively as follows:

$$treeof(\mathsf{Top})(\bullet) \equiv \mathsf{Top}$$

$$treeof(T_1 \to T_2)(\bullet) \equiv \to \qquad treeof(T_1 \to T_2)(i, \pi) \equiv treeof(T_i)(\pi)$$

$$treeof(T_1 \times T_2)(\bullet) \equiv \times \qquad treeof(T_1 \times T_2)(i, \pi) \equiv treeof(T_i)(\pi)$$

$$treeof(\mu X.\, T)(\pi) \equiv treeof([X \mapsto \mu X.\, T]T)(\pi)$$

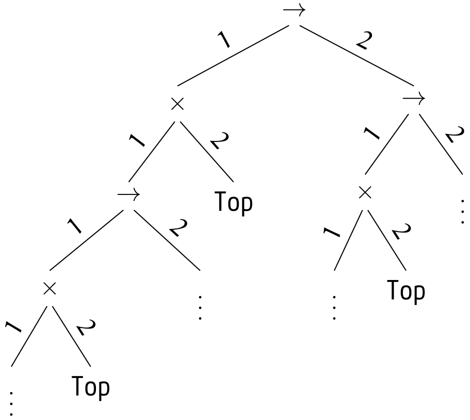
Question

Why is *treeof* well-defined?

Answer

Every recursive use of *treeof* on the right-hand side reduces the lexicographic size of the pair $(|\pi|, \mu\text{-}height(T))$, where $\mu\text{-}height(T)$ is the number of of μ-bindings at the front of T.

$treeof(\mu X.((X \times \mathsf{Top}) \to X))$



Aside: Why is Coinductive Subtyping Indeed Correct?

Subtype Relation

Let $\mathcal{T}$ denote the set of all tree types. Two tree types S and T are said to be in the **subtype relation** ("S is a subtype of T") if $(S, T) \in \nu F$, where the monotone function $F : \mathcal{P}(\mathcal{T} \times \mathcal{T}) \to \mathcal{P}(\mathcal{T} \times \mathcal{T})$ is defined as follows:

$$
\begin{aligned}
F(R) \equiv\ & \{(T, \mathsf{Top}) \mid T \in \mathcal{T}\} \\
& \cup \{(S_1 \to S_2, T_1 \to T_2) \mid (T_1, S_1), (S_2, T_2) \in R\} \\
& \cup \{(S_1 \times S_2, T_1 \times T_2) \mid (S_1, T_1), (S_2, T_2) \in R\}
\end{aligned}
$$

Theorem

Recall that F_d is the generating function for the subtype relation on μ-types.
Let $(S, T) \in \mathcal{T}_m \times \mathcal{T}_m$. Then $(S, T) \in \nu F_d$ **if and only if** $(\mathit{treeof}(S), \mathit{treeof}(T)) \in \nu F$.

Homework

Question

- Implement Y_T (shown on Slide 23) in OCaml/MoonBit. 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 three recursive functions.
- Formulate your solution with explicit `fold`'s and `unfold`'s. You may check your solution using the `fullisorec` checker.