Agdaによる
依存型プログラミング入門

叢 悠悠 (東京科学大学)

PPL サマースクール 2025

自己紹介

- 名前：叢 悠悠 (そう ゆうゆう)
- 経歴：お茶大 PhD → 東工大 / Science Tokyo 助教
- 興味：継続、型、自然言語、音楽

本講演のねらいと流れ

依存型の嬉しさ・面白さを体感しよう

- 依存型と Agda の基本
- 簡単な依存型プログラミング
- 音楽への応用

資料置き場

依存型とは

- 項に依存する型
  - 例：`Vec Int 3` は3つの整数を含むリストの型
- 多くの定理証明支援系が提供

依存型の嬉しさ：正当性・安全性の保証

- 失敗しないリストの参照

```
[a, b, c].4
```

- 整列されたリストを返す並べ替え

```
[2, 3, 1] → [1, 2, 3]
```

性質保証の仕組み：カリー・ハワード対応

論理

命題　$A \to A$

証明　$$\dfrac{[A]_1}{A \to A} \to \mathrm{I}_1$$

プログラミング

型　`A → A`

項
```
id : A → A
id x = x
```

性質を保証するためのアプローチ

- Extrinsic なアプローチ：プログラムと証明が分離

```
sort : List A → List A
sort-correct : (l : List A) → Sorted (sort l)
```

- Intrinsic なアプローチ：プログラムと証明が一体化

```
sort : List A → SortedList A
```

本講演

Agda プログラムの構成要素

Double.agda

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))
```

Agda プログラムの構成要素

Double.agda

```
module Double where
```

モジュール宣言

```
open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))
```

Agda プログラムの構成要素

```
module Double where
```

モジュール名・
ファイル名は同一

Double.agda

```
open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))
```

Agda プログラムの構成要素

Double.agda

```
module Double where
```

モジュール読み込み

```
open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (
```

標準ライブラリ

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Agda プログラムの構成要素

Double.agda

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))
```

型定義, 関数定義, etc.

Agda の型定義

型の名前　　型の型

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

コンストラクタ

Agda の関数定義

関数名　　　入出力型

```
double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))
```

本体

Unicode の入力

シンボル	VS Code	Emacs
ℕ	`]bN`	`\bN`
λ	`]Gl`	`\Gl`
∀	`]all`	`\all`
→	`]r-`	`\r-`
x_1	`x]_1`	`x\_1`

キーバインディング

キー	操作
C-c C-l	読み込み
C-c C-n	正規化 (評価)
C-c C-,	ゴール・環境の確認
C-c C-c	場合分け
C-c C-r	穴埋め

* C = control キー

プログラムの読み込み (C-c C-l)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))
```

プログラムの読み込み (C-c C-l)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))

```

Agda

All Done

AGDA v2.6.4.1

プログラムの正規化 (C-c C-n)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))

```

Agda

Compute normal form (DefaultCompute)

AGDA v2.6.4.1

expression to normalize:

プログラムの正規化 (C-c C-n)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))

```

Agda

Compute normal form (DefaultCompute) AGDA v2.6.4.1

double 3

プログラムの正規化 (C-c C-n)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))

```

Agda

Normal form AGDA v2.6.4.1

```
6
```

穴の記述 (?)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double n = ?

```

Agda

Normal form

AGDA V2.6.4.1

6

穴の記述 (?)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double n = {!   0!}

```

Agda

All Goals AGDA v2.6.4.1

```
?0 : ℕ
```

ゴールと環境の確認 (C-c C-,)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double n = {!  0!}

```

Agda

All Goals

Agda v2.6.4.1

```
?0 : ℕ
```

ゴールと環境の確認 (C-c C-,)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double n = {!  0!}

```

Agda

Goal and Context AGDA v2.6.4.1

ℕ GOAL

n : ℕ

場合分け (C-c C-c)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double n = {!   0!}

```

Agda

Case

Agda v2.6.4.1

Please specify which variable(s) you wish to split, multiple variables are delimited by whitespaces

variable(s) to case split:

場合分け (C-c C-c)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double n = {!   0!}

```

Agda

Case

Agda v2.6.4.1

Please specify which variable(s) you wish to split, multiple variables are delimited by whitespaces

n

場合分け (C-c C-c)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero = {!   0!}
double (suc n) = {!   1!}
```

Agda

All Goals　AGDA v2.6.4.1

```
?0 : ℕ
?1 : ℕ
```

穴埋め (C-c C-r)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero = {! zero  0!}
double (suc n) = {!   1!}

```

Agda

All Goals   Agda v2.6.4.1

```
?0 : ℕ
?1 : ℕ
```

穴埋め (C-c C-r)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero = zero
double (suc n) = {!   1!}

```

Agda

All Goals

AGDA v2.6.4.1

```
?1 : ℕ
```

穴埋め (C-c C-r)

```
module Double where

open import Data.Nat using (ℕ; zero; suc)

double : ℕ → ℕ
double zero = zero
double (suc n) = {! x  1!}

```

Agda

All Goals AGDA v2.6.4.1

```
?1 : ℕ
```

穴埋め (C-c C-r)

```
1  module Double where
2
3  open import Data.Nat using (ℕ; zero; suc)
4
5  double : ℕ → ℕ
6  double zero = zero
7  double (suc n) = {! x  1!}
8
```

Agda ×

Error AGDA V2.6.4.1

```
/Users/youyoucong/pplss-2025/Double.agda:7,20-21
Not in scope:
  x at
/Users/youyoucong/pplss-2025/Double.agda:7,20-21
when scope checking x
```

ERROR

依存型プログラミング　入門編：

リスト操作の性質を保証しよう

通常のリスト

標準ライブラリ

```
data List (A : Set) : Set where
  []  : List A
  _∷_ : A → List A → List A
```

通常のリスト

標準ライブラリ

```
data List (A : Set) : Set where
  []  : List A
  _∷_ : A → List A → List A
```

要素の型
(パラメータ)

通常のリスト

標準ライブラリ

```
data List (A : Set) : Set where
  []  : List A
  _∷_ : A → List A → List A
```

1文字（]:: or \::）

通常のリスト

標準ライブラリ

```
data List (A : Set) : Set where
  []  : List A
  _∷_ : A → List A → List A
```

引数の位置

長さの情報を持つリスト

標準ライブラリ

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _∷_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

長さの情報を持つリスト

標準ライブラリ

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _∷_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

リストの長さ
（インデックス）

長さの情報を持つリスト

標準ライブラリ

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero                  空リストの長さは0
  _∷_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

長さの情報を持つリスト

標準ライブラリ

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _∷_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

暗黙の引数 (Agda が推論)

長さの情報を持つリスト

標準ライブラリ

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _∷_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

長さ n のリストに要素を1つ追加した結果は長さ suc n

List 上の append 関数

```
appendL : List A → List A → List A
appendL []       l = l
appendL (x ∷ xs) l = x ∷ appendL xs l
```

Vec 上の append 関数

```
appendV : Vec A m → Vec A n → Vec A (m + n)
appendV []       l = l
appendV (x ∷ xs) l = x ∷ appendV xs l
```

Vec 上の append 関数

```
appendV : Vec A m → Vec A n → Vec A (m + n)
appendV []       l = l
appendV (x ∷ xs) l = x ∷ appendV xs l
```

型から得られる保証：
appendV の結果の要素数は入力リストの要素数の和

演習問題 1

A → B 型の関数 f と Vec A n 型のリスト l を受け取り、
l の各要素に f を適用する関数 mapV 関数を定義せよ。
この関数の型はどのような性質を保証するだろうか。

参考：List 上の map 関数

```
mapL : (A → B) → List A → List B
mapL f []        = []
mapL f (x ∷ xs) = f x ∷ mapL f xs
```

List 上の head 関数

```
headL : List A → A
headL l = ?
```

List 上の head 関数

```
headL : List A → A
headL []       = ?
headL (x ∷ xs) = ?
```

List 上の head 関数

```
headL : List A → A
headL []          = ?
headL (x ∷ xs) = x
```

List 上の head 関数

```
headL : List A → A
headL (x ∷ xs) = x
```

← 実行不可能

```
Incomplete pattern matching for headL.
Missing cases:
  headL []
when checking the definition of headL
```

List 上の head 関数

```
headL : List A → Maybe A
headL []           = nothing
headL (x ∷ xs) = just x
```

Vec 上の head 関数

```
headV : Vec A (suc n) → A
headV l = ?
```

Vec 上の head 関数

```
headV : Vec A (suc n) → A
headV l = ?
```

長さが `suc n` ⇒ 空でない
(cf. `[] : Vec A zero`)

Vec 上の head 関数

```
headV : Vec A (suc n) → A
headV (x ∷ xs) = ?
```

Vec 上の head 関数

```
headV : Vec A (suc n) → A
headV (x ∷ xs) = x
```

Vec 上の head 関数

```
headV : Vec A (suc n) → A
headV []          = ?
headV (x ∷ xs) = x
```

The case for the constructor [] is impossible
because unification ended with
a conflicting equation
  zero ≟ suc n

List 上の lookup 関数

```
lookupL : List A → ℕ → A
lookupL []       n       = ?
lookupL (x ∷ xs) zero    = x
lookupL (x ∷ xs) (suc n) = lookupL xs n
```

List 上の lookup 関数

```
lookupL : List A → ℕ → A           リストの長さ未満
lookupL []         n       = ?
lookupL (x ∷ xs) zero    = x
lookupL (x ∷ xs) (suc n) = lookupL xs n
```

有限集合 Fin

Fin n =
n 未満の自然数

標準ライブラリ

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Fin 0 型の値：

Fin 1 型の値：
zero

Fin 2 型の値：
zero
suc zero

Vec 上の lookup 関数

```
lookupV : Vec A n → Fin n → A
lookupV (x ∷ xs) zero    = x
lookupV (x ∷ xs) (suc n) = lookupV xs n
```

Vec 上の lookup 関数

```
lookupV : Vec A n → Fin n → A    リストの長さ未満
lookupV (x ∷ xs) zero     = x
lookupV (x ∷ xs) (suc n) = lookupV xs n
```

List 上の lookup 関数 (別解)

```
lookupL2 : (l : List A) → Fin (length l) → A
lookupL2 (x ∷ xs) zero    = x
lookupL2 (x ∷ xs) (suc n) = lookupL2 xs n
```

List 上の lookup 関数 (別解)

```
lookupL2 : (l : List A) → Fin (length l) → A
lookupL2 (x ∷ xs) zero    = x  リストの長さ未満
lookupL2 (x ∷ xs) (suc n) = lookupL2 xs n
```

発展

- 行列の表現
  - 型を行数と列数の情報でインデックス
- 安全な行列演算
  - 型から得られる情報を用いて次元の不一致を検知

依存型プログラミング　初級編：

型付き言語・インタプリタを実装しよう

BoolNat 言語

- 真偽値　`true`, `false`
- 自然数　`0`, `1`, `2`, …
- 条件文　`if` e_1 `then` e_2 `else` e_3

BoolNat 言語の式 (型なし)

```
data Exp : Set where
  tru  : Exp
  fls  : Exp
  num  : ℕ → Exp
  ifte : Exp → Exp → Exp → Exp
```

型なしの式の例

```
exp1 : Exp  -- if true then 1 else 0
exp1 = ifte tru (num 1) (num 0)

exp2 : Exp  -- if 1 then 1 else 0
exp2 = ifte (num 1) (num 1) (num 0)

exp3 : Exp  -- if true then 1 else false
exp3 = ifte tru (num 1) fls
```

BoolNat 言語の値 (型なし)

```
data Val : Set where
  vtru : Val
  vfls : Val
  vnum : ℕ → Val
```

型なし言語のインタプリタ

```
interp : Exp → Val
interp tru      = vtru
interp fls      = vfls
interp (num n) = vnum n
interp (ifte e₁ e₂ e₃) with interp e₁ =
... | vtru   = interp e₂
... | vfls   = interp e₃
... | vnum n = ?
```

BoolNat 言語の型

```
data Ty : Set where
  boolty : Ty
  numty  : Ty
```

BoolNat 言語の式 (型付き)

```
data TExp : Ty → Set where
  tru  : TExp boolty
  fls  : TExp boolty
  num  : ℕ → TExp numty
  ifte : {τ : Ty} →
         TExp boolty → TExp τ → TExp τ → TExp τ
```

BoolNat 言語の式 (型付き)

```
data TExp : Ty → Set where
  tru  : TExp boolty
  fls  : TExp boolty
  num  : ℕ → TExp numty
  ifte : {τ : Ty} →
         TExp boolty → TExp τ → TExp τ → TExp τ
```

tru, fls は
真偽値型の式

BoolNat 言語の式 (型付き)

```
data TExp : Ty → Set where
  tru   : TExp boolty
  fls   : TExp boolty
  num   : ℕ → TExp numty
  ifte : {τ : Ty} →
          TExp boolty → TExp τ → TExp τ → TExp τ
```

num n は
自然数型の式

BoolNat 言語の式 (型付き)

```
data TExp : Ty → Set where
  tru  : TExp boolty
  fls  : TExp boolty
  num  : ℕ → TExp numty
  ifte : {τ : Ty} →
         TExp boolty → TExp τ → TExp τ → TExp τ
```

条件部分は真偽値型の式

BoolNat 言語の式 (型付き)

```
data TExp : Ty → Set where
  tru  : TExp boolty
  fls  : TExp boolty
  num  : ℕ → TExp numty
  ifte : {τ : Ty} →
         TExp boolty → TExp τ → TExp τ → TExp τ
```

then 節, else 節は同じ型の式

型が付く式の例

```
texp1 : TExp numty   -- if true then 1 else 0
texp1 = ifte tru (num 1) (num 0)
```

✓ 型検査通過

型が付かない式の例

```
texp2 : TExp numty  -- if 1 then 1 else 0
texp2 = ifte (num 1) (num 1) (num 0)
```

❌型エラー

```
             TExp numty
```

型が付かない式の例

```
texp3 : TExp numty   -- if true then 1 else false
texp3 = ifte tru (num 1) fls            ✗型エラー
                 TExp numty  TExp boolty
```

BoolNat 言語の値 (型付き)

```
data TVal : Ty → Set where
  vtru : TVal boolty
  vfls : TVal boolty
  vnum : ℕ → TVal numty
```

型付き言語のインタプリタ

```
interp2 : TExp τ → TVal τ
interp2 tru      = vtru
interp2 fls      = vfls
interp2 (num n) = vnum n
interp2 (ifte e₁ e₂ e₃) with interp2 e₁ =
... | vtru = interp2 e₂
... | vfls = interp2 e₃
```

型付き言語のインタプリタ

```
interp2 : TExp τ → TVal τ
interp2 tru        = vtru
interp2 fls        = vfls
interp2 (num n) = vnum n
interp2 (ifte e₁ e₂ e₃) with interp2 e₁ =
... | vtru = interp2 e₂
... | vfls = interp2 e₃
```

型から得られる保証：
interp2 は型を保存

型付き言語のインタプリタ

```
interp2 : TExp τ → TVal τ
interp2 tru      = vtru
interp2 fls      = vfls
interp2 (num n) = vnum n
interp2 (ifte e₁ e₂ e₃) with interp2 e₁ =
... | vtru = interp2 e₂
... | vfls = interp2 e₃
```

vnum n のケースなし
(interp2 e_1 : TVal boolty)

型の解釈

```
interpTy : Ty → Set
interpTy boolty = Bool
interpTy numty  = ℕ
```

プリミティブの値を返すインタプリタ

```
interp3 : TExp τ → interpTy τ
interp3 tru      = true
interp3 fls      = false
interp3 (num n) = n
interp3 (ifte e₁ e₂ e₃) with interp3 e₁
... | true  = interp3 e₂
... | false = interp3 e₃
```

演習問題２

`TExp` を `is0 e` という形の式で拡張し、`interp2` と `interp3` をそれに合わせて拡張せよ。ただし、`is0` は引数が 0 と等しいかどうかを返す演算子である。

発展

- 単純型付きラムダ計算
  - 式の型を型環境でインデックス
- 依存型付きラムダ計算
  - 型規則と型の同値性規則を同時に定義
    (Altenkirch & Kaposi '16)

依存型プログラミング　中級編：

挿入ソートの正しさを保証しよう

一般のリスト

```
data List (A : Set) : Set where
  []  : List A
  _∷_ : A → List A → List A
```

昇順に整列されたリスト

```
data OList (A : Set) : Set where
  []  : OList A
  _∷_ : (a : A) → OList A → OList A
```

a 以上の要素が昇順に並んだリスト

昇順に整列された自然数のリスト

```
data OList (b : ℕ) : Set where
  nil  : OList b
  cons : (n : ℕ) → b ≤ n → OList n → OList b
```

昇順に整列された自然数のリスト

```
data OList (b : ℕ) : Set where        要素の下限
  nil  : OList b
  cons : (n : ℕ) → b ≤ n → OList n → OList b
```

昇順に整列された自然数のリスト

$$\overbrace{\boxed{n} :: \boxed{n\text{ 以上}}}^{b\text{ 以上}}$$

```
data OList (b : ℕ) : Set where
  nil  : OList b
  cons : (n : ℕ) → b ≤ n → OList n → OList b
```

昇順に整列された自然数のリスト

```
data OList (b : ℕ) : Set where
  nil  : OList b
  cons : (n : ℕ) → b ≤ n → OList n → OList b
```

b は n 以下

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : zero ≤ n
  s≤s : m ≤ n → suc m ≤ suc n
```

標準ライブラリ

整列されたリストの例

```
goodList : OList 0
goodList = cons 1 z≤n (cons 2 (s≤s z≤n) nil)
```

```
data OList (b : ℕ) : Set where
  nil  : OList b
  cons : (n : ℕ) → b ≤ n → OList n → OList b
```

整列されたリストの例

```
goodList : OList 0
goodList = cons 1 z≤n (cons 2 (s≤s z≤n) nil)
```

$$\texttt{goodList = cons }\boxed{\texttt{1}}\texttt{ z≤n }\underbrace{\texttt{(cons 2 (s≤s z≤n) nil)}}_{\texttt{OList 0}}$$

0 ≤ 1

```
data OList (b : ℕ) : Set where
  nil  : OList b
  cons : (n : ℕ) → b ≤ n → OList n → OList b
```

整列されたリストの例

```
goodList : OList 0                    OList 1
goodList = cons 1 z≤n (cons 2 (s≤s z≤n) nil)
                                   1 ≤ 2
```

```
data OList (b : ℕ) : Set where
  nil  : OList b
  cons : (n : ℕ) → b ≤ n → OList n → OList b
```

整列されていないリストの例

```
badList : OList 0            OList 2
badList = cons 2 z≤n (cons 1 ? nil)
                             2 ≤ 1
```

```
data OList (b : ℕ) : Set where
  nil  : OList b
  cons : (n : ℕ) → b ≤ n → OList n → OList b
```

要素の挿入

```
insert : {b : ℕ} →
         (n : ℕ) → b ≤ n → OList b → OList b
insert n b≤n nil = cons n b≤n nil
insert m b≤m (cons n b≤n l) with compare m n
... | less    .m k = cons m b≤m (cons n ≤-suc+ l)
... | equal   .m   = cons m b≤m (cons n ≤-refl l)
... | greater .n k = cons n b≤n (insert m ≤-suc+ l)
```

要素の挿入

insert の入力・出力は
整列されたリスト

```
insert : {b : ℕ} →
         (n : ℕ) → b ≤ n → OList b → OList b
insert n b≤n nil = cons n b≤n nil
insert m b≤m (cons n b≤n l) with compare m n
... | less    .m k = cons m b≤m (cons n ≤-suc+ l)
... | equal   .m   = cons m b≤m (cons n ≤-refl l)
... | greater .n k = cons n b≤n (insert m ≤-suc+ l)
```

要素の挿入

挿入する要素は
入力・出力の下限以上

```
insert : {b : ℕ} →
         (n : ℕ) → b ≤ n → OList b → OList b
insert n b≤n nil = cons n b≤n nil
insert m b≤m (cons n b≤n l) with compare m n
... | less    .m k = cons m b≤m (cons n ≤-suc+ l)
... | equal   .m   = cons m b≤m (cons n ≤-refl l)
... | greater .n k = cons n b≤n (insert m ≤-suc+ l)
```

要素の挿入

```
insert : {b : ℕ} →
         (n : ℕ) → b ≤ n → OList b → OList b
insert n b≤n nil = cons n b≤n nil
insert m b≤m (cons n b≤n l) with compare m n
... | less    .m k = cons m b≤m (cons n ≤-suc+ l)
... | equal   .m   = cons m b≤m (cons n ≤-refl l)
... | greater .n k = cons n b≤n (insert m ≤-suc+ l)
```

要素の挿入

```
insert : {b : ℕ} →
         (n : ℕ) → b ≤ n → OList b → OList b
insert n b≤n nil = cons n b≤n nil
insert m b≤m (cons n b≤n l) with compare m n
... | less    .m k = cons m b≤m
... | equal   .m   = cons m b≤m (cons n ≤-refl l)
... | greater .n k = cons n b≤n (insert m ≤-suc+ l)
```

m と n の比較

要素の挿入

```
insert : {b : ℕ} →
          (n : ℕ) → b ≤ n → OList b → OList b
insert n b≤n nil = cons n b≤n nil
insert m b≤m (cons n b≤n l) with compare m n
... | less     .m k = cons m b≤m (cons n ≤-suc+ l)
```

n = suc (m + k) の場合　　m ≤ suc (m + k) の証明

```
... | greater .n k = cons n b≤n (insert m ≤-suc+ l)
```

要素の挿入

```
insert : {b : ℕ} →
         (n : ℕ) → b ≤ n → OList b → OList b
insert n b≤n nil = cons n b≤n nil
insert m b≤m (cons n b≤n l) with compare m n
... | less    .m k = cons m b≤m (cons n ≤-suc+ l)
... | equal   .m   = cons m b≤m (cons n ≤-refl l)
```

n = m の場合

m ≤ m の証明

要素の挿入

```
insert : {b : ℕ} →
         (n : ℕ) → b ≤ n → OList b → OList b
insert n b≤n nil = cons n b≤n nil
insert m b≤m (cons n b≤n l) with compare m n
... | less    .m k = cons m b≤m (cons n ≤-suc+ l)
```

m = suc (n + k) の場合　　n ≤ suc (n + k) の証明

```
... | greater .n k = cons n b≤n (insert m ≤-suc+ l)
```

挿入ソート

```
isort : List ℕ → OList 0
isort []       = nil
isort (n ∷ l) = insert n z≤n (isort l)
```

挿入ソート

```
isort : List ℕ → OList 0
isort []       = nil
isort (n ∷ l) = insert n z≤n (isort l)
```

isort の結果は
整列されたリスト

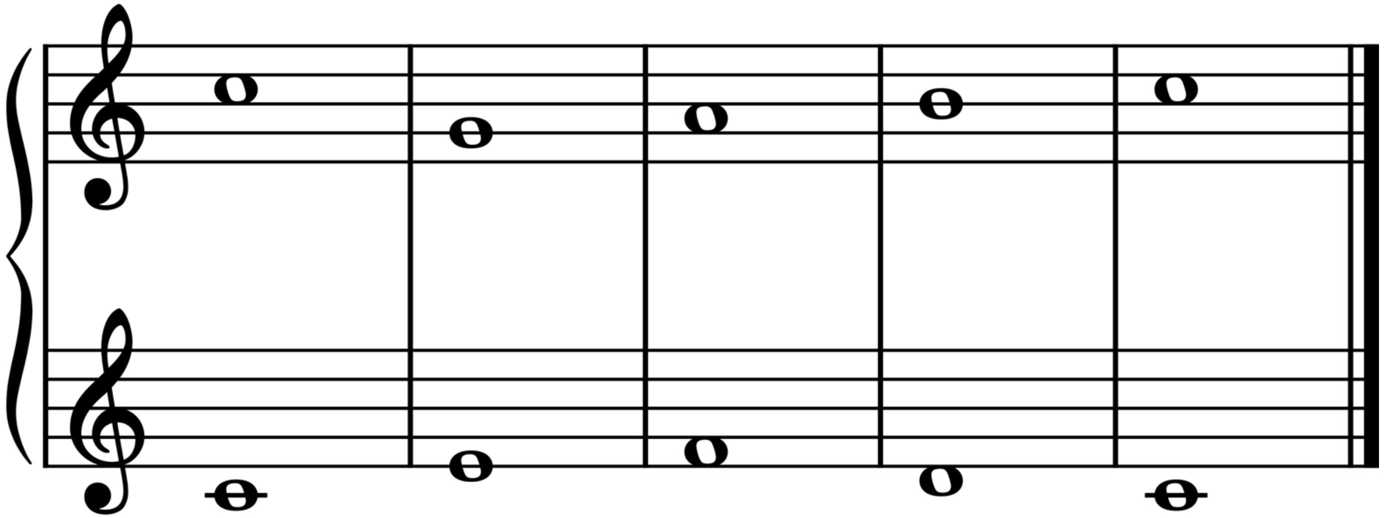
発展

- ソート前後の要素の保存
  - 入力・出力リストが並べ替えの関係であることを型として表現
- 二分探索木の表現
  - 要素の下限・上限の情報を型に持たせる

依存型プログラミング　応用編：

音楽の正しさを保証しよう

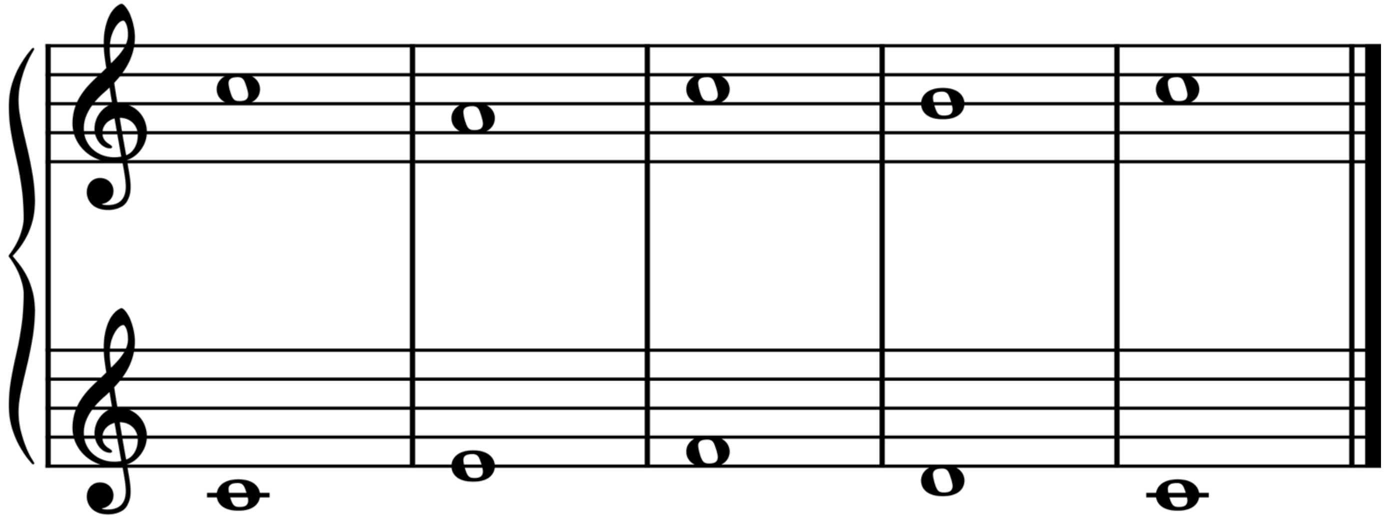
対位法による旋律の重ね合わせ (良い例)



対旋律
(counterpoint)

定旋律
(cantus firmus)

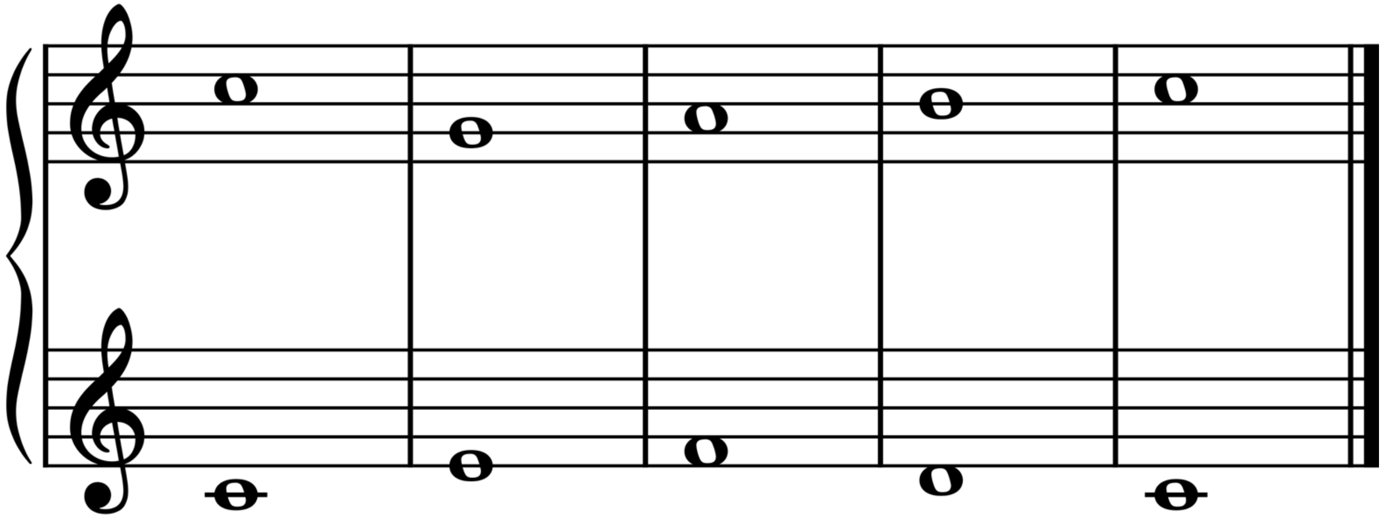
対位法による旋律の重ね合わせ (悪い例)



対旋律
(counterpoint)

定旋律
(cantus firmus)

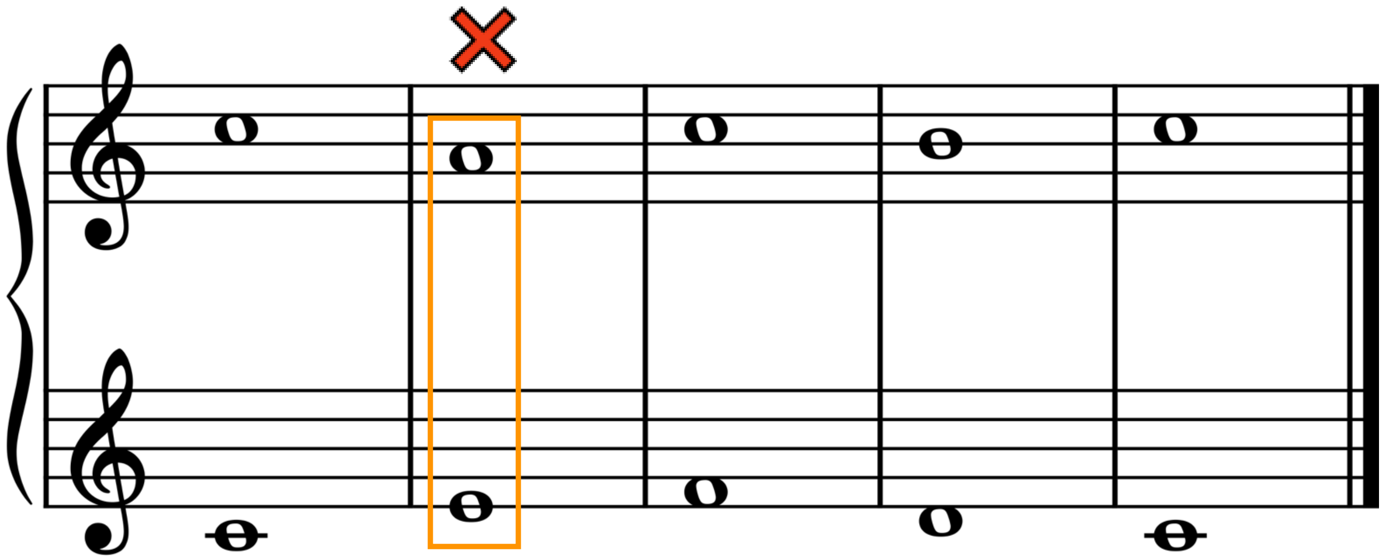
対位法の規則 ①：すべて協和音



8度　3度　3度　6度　8度

協和音 ＝
1度, 3度, 5度,
6度, 8度

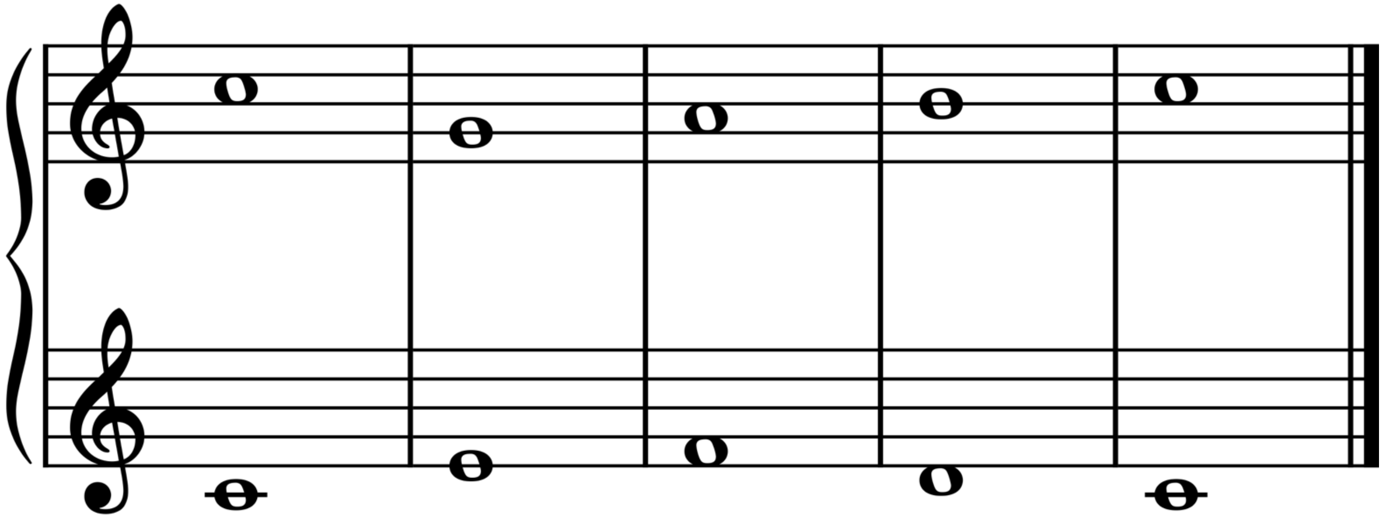
対位法の規則 ①：すべて協和音



8度　4度　5度　6度　8度

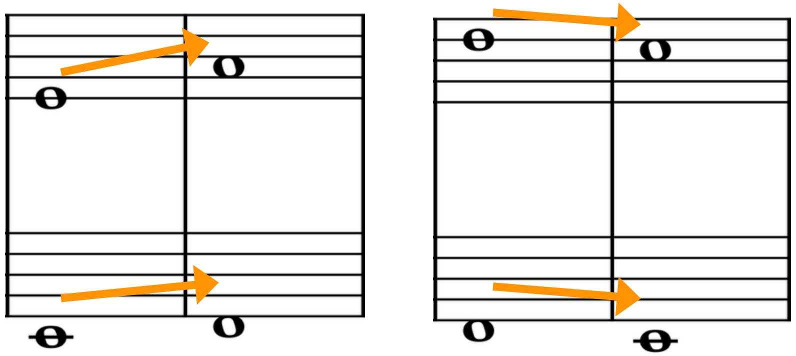
協和音 ＝
1度, 3度, 5度,
6度, 8度

対位法の規則 ②：並達5度・8度なし

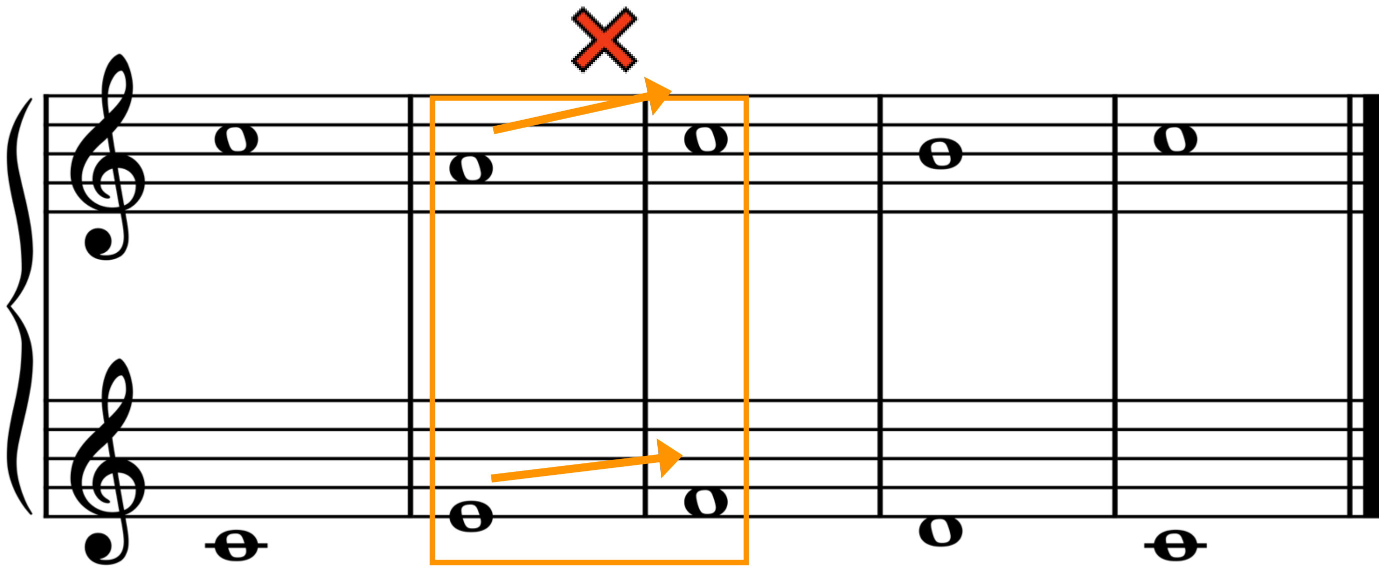


8度　3度　3度　6度　8度

並達5度・8度 ＝

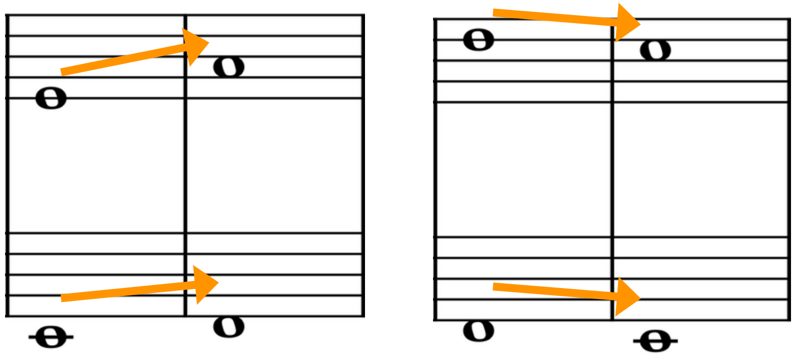


対位法の規則 ②：並達5度・8度なし

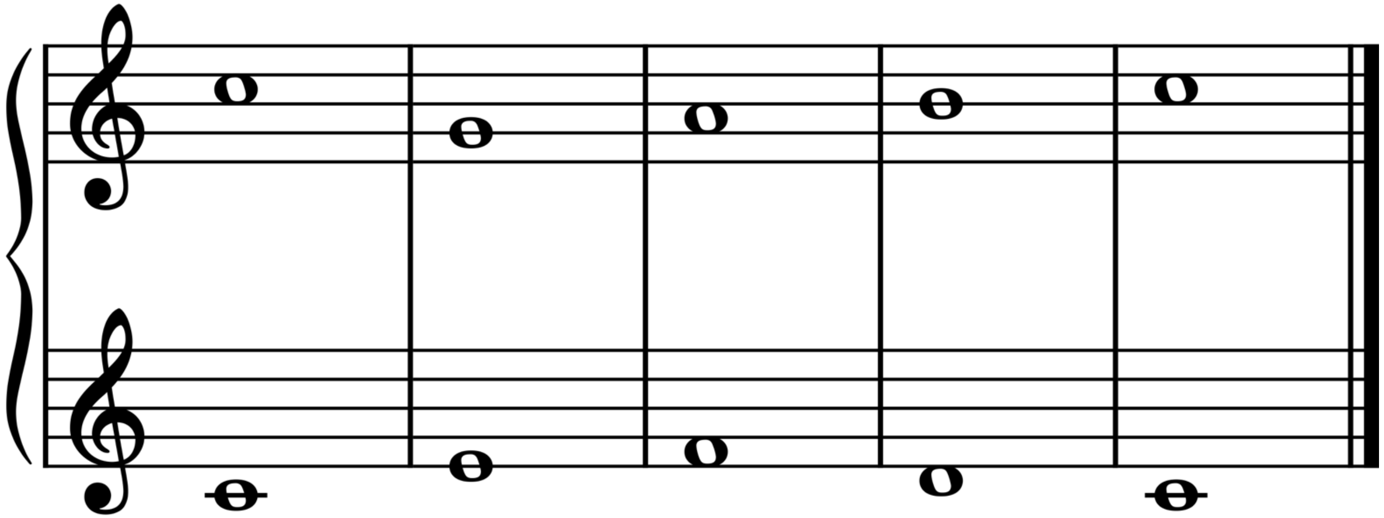


8度　4度　5度　6度　8度

並達5度・8度 ＝



音楽の表現 (イメージ)



```
Bar =
Pitch × Interval

  [(c , oct),
   (e , 3rd),
   (f , 3rd),
   (d , 6th),
   (c , oct)]
```

ピッチとインターバル

```
data Pitch : Set where
  c  : Pitch
  d  : Pitch
  e  : Pitch
  f  : Pitch
  g  : Pitch
  a  : Pitch
  b  : Pitch
  c2 : Pitch
```

```
data Interval : Set where
  uni : Interval
  2nd : Interval
  3rd : Interval
  4th : Interval
  5th : Interval
  6th : Interval
  7th : Interval
  oct : Interval
```

小節

```
Bar : Set
Bar = Pitch × Interval

record _×_ : (A B : Set) → Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B
```

標準ライブラリ

, の前後にスペースが必要

対位法に従った音楽

- ベースケースは要素数1のリスト
- 要素は後ろに追加

```
data CP where
  first  : (bar : Bar) → valid1 bar → CP
  extend : (cp : CP) → (bar : Bar) →
           valid1 bar →
           valid2 (last cp) bar →
           CP
```

対位法に従った音楽

1小節の正しさ

```
data CP where
  first  : (bar : Bar) → valid1 bar → CP
  extend : (cp : CP) → (bar : Bar) →
           valid1 bar →
           valid2 (last cp) bar →
           CP
```

対位法に従った音楽

```
data CP where
  first  : (bar : Bar) → valid1 bar → CP
  extend : (cp : CP) → (bar : Bar) →
           valid1 bar →
           valid2 (last cp) bar →
           CP
```

最後の1小節の正しさ

対位法に従った音楽

```
data CP where
  first  : (bar : Bar) → valid1 bar → CP
  extend : (cp : CP) → (bar : Bar) →
           valid1 bar →
           valid2 (last cp) bar →
           CP
```

最後の2小節
の正しさ

1つの小節に対する規則

```
valid1 : Bar → Set
valid1 bar = Consonant (proj₂ bar)
```

協和音である

1つの小節に対する規則

```
valid1 : Bar → Set
valid1 bar = Consonant (proj₂ bar)
```

bar は協和音

和音が協和音であることの証明

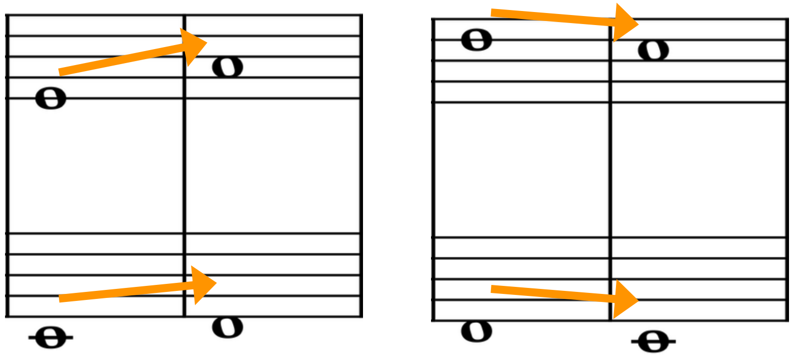
```
data Consonant : Interval → Set where
  uniIsConsonant : Consonant uni
  3rdIsConsonant : Consonant 3rd
  5thIsConsonant : Consonant 5th
  6thIsConsonant : Consonant 6th
  octIsConsonant : Consonant oct
```

隣接する2小節に対する規則

```
valid2 : Bar → Bar → Set
valid2 bar1 bar2 =
  Not58 (proj₂ bar2) ⊎ NotDirect bar1 bar2
```

並達5度・8度でない

復習：並達5度・8度

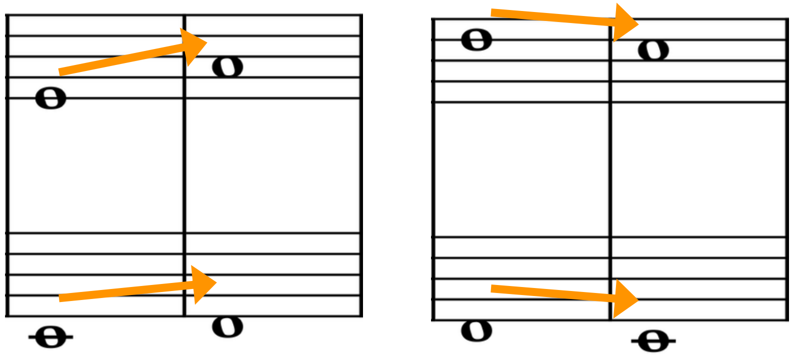


隣接する2小節に対する規則

```
valid2 : Bar → Bar → Set
valid2 bar1 bar2 =
  Not58 (proj₂ bar2) ⊎ NotDirect bar1 bar2
```

2つ目の和音は5度・8度以外

復習：並達5度・8度



隣接する2小節に対する規則

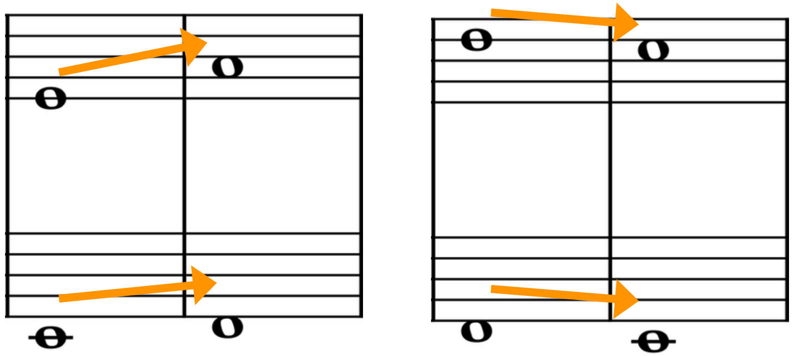
valid2 : Bar → Bar → Set

valid2 bar1 bar2 =

  Not58 (proj_2 bar2) ⊎ NotDirect bar1 bar2

2声の進行方向が同じでない

復習：並達5度・8度



隣接する2小節に対する規則

```
valid2 : Bar → Bar → Set
valid2 bar1 bar2 =
  Not58 (proj₂ bar2) ⊎ NotDirect bar1 bar2
```

論理和

```
data ⊎ : Set → Set → Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

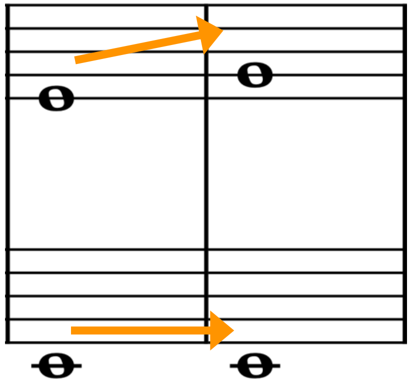
標準ライブラリ

和音が5度・8度以外であることの証明

```
data Not58 : Interval → Set where
  uniIsNot58 : Not58 uni
  2ndIsNot58 : Not58 2nd
  3rdIsNot58 : Not58 3rd
  4thIsNot58 : Not58 4th
  6thIsNot58 : Not58 6th
  7thIsNot58 : Not58 7th
```

進行方向が同じでないことの証明 (抜粋)

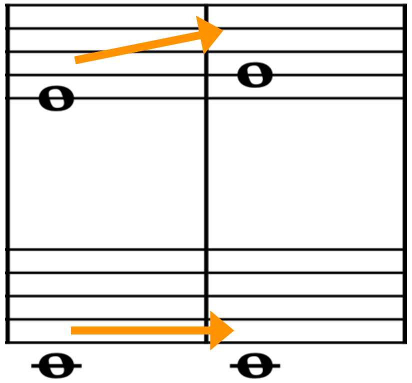
```
data NotDirect (bar1 bar2 : Bar) : Set where
  oblique1 :
    dir (cf bar1) (cf bar2) ≡ stay →
    NotDirect bar1 bar2
```



進行方向が同じでないことの証明 (抜粋)

```
data NotDirect (bar1 bar2 : Bar) : Set where
  oblique1 :
    dir (cf bar1) (cf bar2) ≡ stay →
    NotDirect bar1 bar2
```

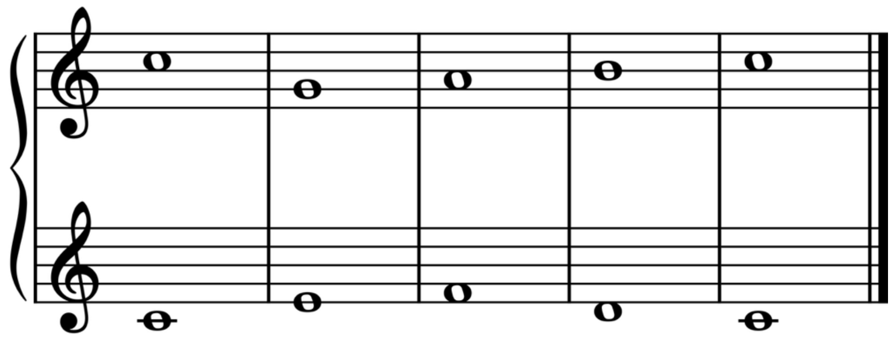
同値



標準ライブラリ

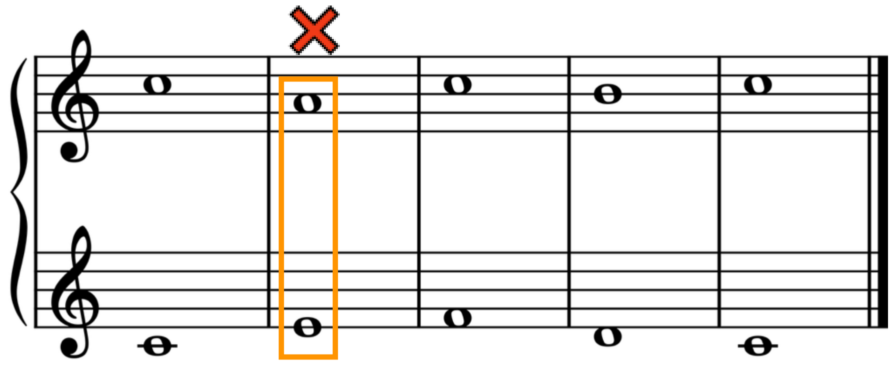
```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x
```

規則を守った音楽の例



```
extend
  (extend
    (extend
      (extend (first (c , oct) octIsConsonant)
              (e , 3rd) 3rdIsConsonant (inj₁ 3rdIsNot58))
      (f , 3rd) 3rdIsConsonant (inj₁ 3rdIsNot58))
    (d , 6th) 6thIsConsonant (inj₁ 6thIsNot58))
  (c , oct) octIsConsonant (inj₂ (contrary2 (refl , refl)))
```

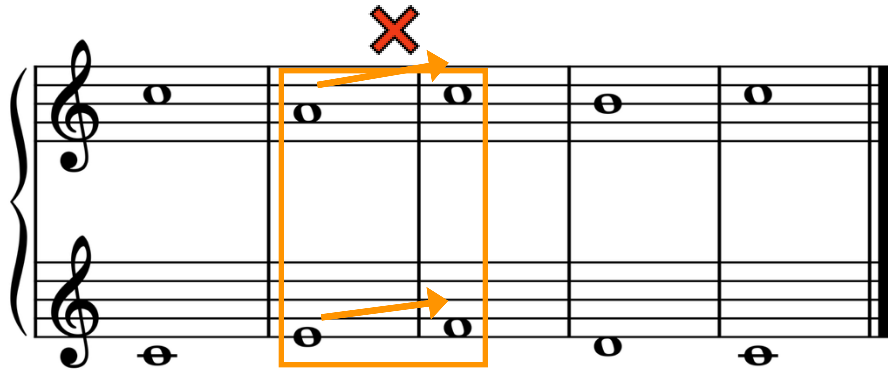
規則を破った音楽の例



```
extend
  (extend
    (extend
      (extend (first (c , oct) octIsConsonant)
              (e , 4th) ? (inj₁ 4thIsNot58))
      (f , 5th) 5thIsConsonant ?)
    (d , 6th) 6thIsConsonant (inj₁ 6thIsNot58))
  (c , oct) octIsConsonant (inj₂ (contrary2 (refl , refl)))
```

Consonant 4th
が必要

規則を破った音楽の例



```
extend
  (extend
    (extend
      (extend (first (c , oct) octIsConsonant)
              (e , 4th) ? (inj₁ 4thIsNot58))
      (f , 5th) 5thIsConsonant ?)
    (d , 6th) 6thIsConsonant (inj₁ 6thIsNot58))
  (c , oct) octIsConsonant (inj₂ (contrary2 (refl , refl)))
```

```
Not58 5th ⊎
NotDirect
  (e , 4th)
  (f , 5th)
```
が必要

演習問題３（オプショナル）

CP 型の音楽を自由に作成し、型が付いた音楽が良い音楽であるかどうかを確かめよ。

cf. "Well-Typed Music Does Not Sound Wrong"
(Szamozvancev & Gale '17)

おわりに

依存型はいろいろな性質の保証に役立つ！

Agda 学習リソース集：

https://agda.readthedocs.io/en/latest/getting-started/tutorial-list.html

Agda Zulip Server: https://agda.zulipchat.com/