Rust 世界の二つのモナド

Rust でも **do**記法をして
プログラムを直感的に記述する件について

konn
関数型まつり 2025
2025-06-14

Slides available: `https://u.konn-san.com/qdo`

自己紹介

- 石井 大海（いしい ひろみ） / konn (Twitter: @mr_konn)
  - いちおう博士（理学）。かつての専門は数理論理学（特に集合論）と計算機代数（Gröbner基底とか）
- 現職：株式会社 Jij ソフトウェア開発チーム（2024/11〜）
  - **Rust** で**数理最適化の埋込ドメイン特化言語 (EDSL) を開発**しています
    - **スポンサーブース出展中！**興味のある方もない方もぜひ！
  - 宣伝：社での取り組みを国際学会 PLDI の併設会議 EGRAPHS で発表する予定
    - なんか雑に日程組んだら明日の深夜に韓国にいくことになってしまい明日は17時で抜けることに（そんな……）
- 前職では **Haskell** で数値計算向け **EDSL** を開発（EDSL 開発に縁があるね）
- 今日の話題は EDSL の設計に欠かせない **do 記法**のお話です

本題

Rust 世界の二つのモナド

Rust でも **do**記法をして

プログラムを直感的に記述する件について

Rust 世界の二つのモナド

Rust でも **do記法**をして

プログラムを直感的に記述する件について

🤔 **モナド**とか **do記法**とかって

Haskeller が使ってるやつ？

🤔 Rust と関係あるの？

二つってなに？

お答えしましょう！

今回の内容

1. `qualified_do`: Rust でも (Qualified)do する件について
   - Haskell の **`QualifiedDo` 記法**を移植した `qualified_do` クレートを紹介
   - **Rust でも do記法は便利そ〜**と思ってもらう
2. do記法でわかる_{かもしれない}！モナド
   - **「do記法が使える構造」**として**モナド**を捉えなおす
   - **Functor** や **Applicative** なども「**do記法の機能をどこまで使えるか**」という観点で説明
   - **GATs** を使って **Rust** で Functor / Applicative / Monad を定式化するトリックも紹介
3. Rust 世界の二つのモナド 〜資源は大切に〜
   - Rust や Linear Haskell など**資源の使い方に厳しい世界**では、**モナド階層が二つに分岐**することを紹介
   - Rust と Linear Haskell では**型システムの違い**によって**何がモナドになれるかに差異**が生まれることを紹介

おしながき

1. **`qualified_do`**: Rust でも (Qualified)do する件について
2. do記法でわかるかもしれない！モナド
3. Rust 世界の二つのモナド 〜資源は大切に〜

おしながき

1. **`qualified_do`**: Rust でも (Qualified)do する件について

`qualified_do`:

Rust でも (Qualified)do する件について

qualified_do クレート

- **qualified_do**: Haskell の QualifiedDo + ApplicativeDo 拡張相当を Rust の proc macro として実装したもの
  - 紹介記事（Zenn）：https://u.konn-san.com/zenn_qdo
  - Crates.io: https://crates.io/crates/qualified_do
- 半年前から仕事で Rust を使うようになり、**do記法が使いたい**！となって開発（業務でも一部利用中）
  - do 記法をサポートする crate は複数あったがどれもメンテナンスされておらず、またdo 記法の実装が**決め打ち**だった

用例1：proptest の Arbitrary

- proptest: Rust の性質ベーステストライブラリ
  - classify 等の統計機能を除けば必要な機能が一通り揃っている
- データ生成の Arbitrary trait は derive できるが、手で書く場合は prop_compose! マクロと .prop_map() や .flat_map() で頑張る

例1：qualified_do + proptest

```rust
fn gen_expr() -> impl Strategy<Value = Expr> {
 let leaf = any().prop_map(Expr::Num).boxed();
  leaf.prop_recursive(8, 256, 10, |inner| {
    prop_oneof![
      (inner.clone(), inner.clone())
        .prop_map(|(l, r)|
          Expr::Add(l.into(), r.into())
        ),
      (inner.clone(), inner.clone())
        .prop_map(|(l, r)|
          Expr::Mul(l.into(), r.into())
        ),
    ]
  })
}
```

例1：qualified_do + proptest

```rust
fn gen_expr() -> impl Strategy<Value = Expr> {
 let leaf = any().prop_map(Expr::Num).boxed();
  leaf.prop_recursive(8, 256, 10, |inner| {
    prop_oneof![
      (inner.clone(), inner.clone())
        .prop_map(|(l, r)|
          Expr::Add(l.into(), r.into())
        ),
      (inner.clone(), inner.clone())
        .prop_map(|(l, r)|
          Expr::Mul(l.into(), r.into())
        ),
    ]
  })
}
```

```rust
fn gen_expr() -> impl Strategy<Value = Expr> {
  use qualified_do::qdo;
  let leaf = any().prop_map(Expr::Num).boxed();
  leaf.prop_recursive(8, 256, 10, |inner| {
    prop_oneof![
      qdo! { BoxedProptest {
        l ← inner.clone();
        r ← inner.clone();
        return Expr::Add(l.into(), r.into())
      }},
      qdo! { BoxedProptest {
        l ← inner.clone();
        r ← inner.clone();
        return Expr::Mul(l.into(), r.into())
      }}
    ]
  })
}
```

🤔見通しはいいけど、要る？

もう少し嬉しい例：？記法の代替

- **？記法**：Rust で Option (Maybe) や Result (Either) など、**失敗し得る計算を途中で短絡評価**できる構文
  - かなりべんり
- **問題点：一つのクロージャにつき一つの型**に対してしか使えない
  - 例：Option と Result を混ぜるには**クロージャを作って即座に消費（IIFE）**する必要
  - 「**いまどのクロージャにいる**のか？」という**大域的な情報**を知らないと、**何に関する？記法**なのかわからない！
- **パタンマッチ**や**条件分岐**も if / match をネストするか .then() を使うことになり見通しがわるい

もう少し嬉しい例：？記法の代替

- **？記法**：Rust で Option (Maybe) や Result (Either) など、**失敗し得る計算を途中で短絡評価**できる構文
  - かなりべんり
- **問題点：一つのクロージャにつき一つの型**に対してしか使えない
  - 例：Option と Result を混ぜるには**クロージャを作って即座に消費（IIFE）**する必要
  - 「**いまどのクロージャにいる**のか？」という**大域的な情報**を知らないと、**何に関する？記法**なのかわからない！
- **パタンマッチ**や**条件分岐**も if / match をネストするか .then() を使うことになり見通しがわるい

```
enum Go { Go, NoGo }
let ans = |go| {
  let x = (|| {
    let x = Some(1i64)?;
    let y = Some(2i64)?;
    if x + y % 2 == 1 {
      if let Go::Go = go {
        return Some(x + y + 100);
      }
    }
    None
  })().ok_or("Fail".to_string());
  let y = Ok(3i64)?;
  Ok(x + y + 1000);
};
```

もう少し嬉しい例：？記法の代替

- **？記法**：Rust で Option (Maybe) や Result (Either) など、**失敗し得る計算を途中で短絡評価**できる構文
  - かなりべんり
- **問題点：一つのクロージャにつき一つの型**に対してしか使えない
  - 例：Option と Result を混ぜるには**クロージャを作って即座に消費（IIFE）**する必要
  - 「**いまどのクロージャにいる**のか？」という**大域的な情報**を知らないと、**何に関する？記法**なのかわからない！
- **パタンマッチ**や**条件分岐**も if / match をネストするか .then() を使うことになり見通しがわるい

```
enum Go { Go, NoGo }
let ans = |go| {
  let x = (|| {
    let x = Some(1i64)?;
    let y = Some(2i64)?;
    if x + y % 2 == 1 {
      if let Go::Go = go {
        return Some(x + y + 100);
      }
    }
    None
  })().ok_or("Fail".to_string());
  let y = Ok(3i64)?;
  Ok(x + y + 1000);
};
```

Option に対する ?

もう少し嬉しい例：？記法の代替

- **？記法**：Rust で Option (Maybe) や Result (Either) など、**失敗し得る計算を途中で短絡評価**できる構文
  - かなりべんり
- **問題点：一つのクロージャにつき一つの型**に対してしか使えない
  - 例：Option と Result を混ぜるには**クロージャを作って即座に消費（IIFE）**する必要
  - 「**いまどのクロージャにいる**のか？」という**大域的な情報**を知らないと、**何に関する？記法**なのかわからない！
- **パタンマッチ**や**条件分岐**も if / match をネストするか .then() を使うことになり見通しがわるい

```
enum Go { Go, NoGo }
let ans = |go| {
  let x = (|| {
    let x = Some(1i64)?;
    let y = Some(2i64)?;
    if x + y % 2 == 1 {
      if let Go::Go = go {
        return Some(x + y + 100);
      }
    }
    None
  })().ok_or("Fail".to_string());
  let y = Ok(3i64)?;
  Ok(x + y + 1000);
};
```

Option に対する ?

Result に対する ?

もう少し嬉しい例：？記法の代替

- **？記法**：Rust で Option (Maybe) や Result (Either) など、**失敗し得る計算を途中で短絡評価**できる構文
  - かなりべんり
- **問題点：一つのクロージャにつき一つの型**に対してしか使えない
  - 例：Option と Result を混ぜるには**クロージャを作って即座に消費（IIFE）**する必要
  - 「**いまどのクロージャにいる**のか？」という**大域的な情報**を知らないと、**何に関する？記法**なのかわからない！
- **パタンマッチ**や**条件分岐**も if / match をネストするか .then() を使うことになり見通しがわるい

```
enum Go { Go, NoGo }
let ans = |go| {
  let x = (|| {
    let x = Some(1i64)?;
    let y = Some(2i64)?;
    if x + y % 2 == 1 {
      if let Go::Go = go {
        return Some(x + y + 100);
      }
    }
    None
  })().ok_or("Fail".to_string());
  let y = Ok(3i64)?;
  Ok(x + y + 1000);
};
```

IIFE作成

Option に対する ?

IIFE呼出

Result に対する ?

もう少し嬉しい例：？記法の代替

- **？記法**：Rust で Option (Maybe) や Result (Either) など、**失敗し得る計算を途中で短絡評価**できる構文
  - かなりべんり
- **問題点：一つのクロージャにつき一つの型**に対してしか使えない
  - 例：Option と Result を混ぜるには**クロージャを作って即座に消費（IIFE）**する必要
  - 「**いまどのクロージャにいる**のか？」という**大域的な情報**を知らないと、**何に関する？記法**なのかわからない！
- **パタンマッチ**や**条件分岐**も if / match をネストするか .then() を使うことになり見通しがわるい

```
enum Go { Go, NoGo }
let ans = |go| {
  let x = (|| {
    let x = Some(1i64)?;
    let y = Some(2i64)?;
    if x + y % 2 == 1 {
      if let Go::Go = go {
        return Some(x + y + 100);
      }
    }
    None
  })().ok_or("Fail".to_string());
  let y = Ok(3i64)?;
  Ok(x + y + 1000);
};
```

IIFE作成

Option に対する ?

適当な条件と……

引数が Go の時だけ成功

IIFE呼出

Result に対する ?

同じものを qdo！で

同じものを qdo! で

```
let ans = |go: Go| {
  qdo! { Resulted {
    x ← qdo!{ Optioned {
      x ← Some(1i64);
      y ← Some(2i64);
      guard x + y % 2 == 1;
      Go::Go ← Some(go);
      return x + y + 100
    }}.ok_or("Failed".to_string());
    y ← Ok(3i64);
    return x + y + 1000
  }}
};
```

同じものを qdo! で

全体は Result

```
let ans = |go: Go| {
  qdo! { Resulted {
    x ← qdo!{ Optioned {
      x ← Some(1i64);
      y ← Some(2i64);
      guard x + y % 2 == 1;
      Go::Go ← Some(go);
      return x + y + 100
    }}.ok_or("Failed".to_string());
    y ← Ok(3i64);
    return x + y + 1000
  }}
};
```

同じものを qdo! で

```
let ans = |go: Go| {
  qdo! { Resulted {
    x ← qdo!{ Optioned {
      x ← Some(1i64);
      y ← Some(2i64);
      guard x + y % 2 == 1;
      Go::Go ← Some(go);
      return x + y + 100
    }}.ok_or("Failed".to_string());
    y ← Ok(3i64);
    return x + y + 1000
  }}
};
```

全体は Result

局所的に Option

同じものを qdo! で

```
let ans = |go: Go| {
  qdo! { Resulted {
    x ← qdo!{ Optioned {
      x ← Some(1i64);
      y ← Some(2i64);
      guard x + y % 2 == 1;
      Go::Go ← Some(go);
      return x + y + 100
    }}.ok_or("Failed".to_string());
    y ← Ok(3i64);
    return x + y + 1000
  }}
};
```

全体は Result

局所的に Option

複数の型が共存！※1

※1: {} で区切られてはいるじゃん、() を節約できるだけでは？と言われればそうですが、わざわざ IIFE つくるよりは精神衛生上いいと思ってます

同じものを qdo！ で

```
let ans = |go: Go| {
  qdo! { Resulted {
    x ← qdo!{ Optioned {
      x ← Some(1i64);
      y ← Some(2i64);
      guard x + y % 2 == 1;
      Go::Go ← Some(go);
      return x + y + 100
    }}.ok_or("Failed".to_string());
    y ← Ok(3i64);
    return x + y + 1000
  }}
};
```

全体は Result

適当な条件と……

引数が Go の時だけ成功

局所的に Option

複数の型が共存！※1

※1: {} で区切られてはいるじゃん、() を節約できるだけでは？と言われればそうですが、わざわざ IIFE つくるよりは精神衛生上いいと思ってます

同じものを qdo! で

```
let ans = |go: Go| {
  qdo! { Resulted {
    x ← qdo!{ Optioned {
      x ← Some(1i64);
      y ← Some(2i64);
      guard x + y % 2 == 1;
      Go::Go ← Some(go);
      return x + y + 100
    }}.ok_or("Failed".to_string());
    y ← Ok(3i64);
    return x + y + 1000
  }}
};
```

全体は Result

適当な条件と……

引数が Go の時だけ成功

局所的に Option

条件分岐も
束縛として表現！

複数の型が
共存！※1

※1: {} で区切られてはいるじゃん、() を節約できるだけでは？と言われればそうですが、わざわざ IIFE つくるよりは精神衛生上いいと思ってます

例3：Iterator の合成

- 例： **is 内の Someの中身** と、**js内のRightの偶数**全ての組合せの和を取れ

```
let is = vec![Some(1i64), None, Some(3)];
let js = vec![Left(4i64), Right(5), Right(6)];
```

例3：Iterator の合成

- 例： **is 内の Someの中身** と、**js内のRightの偶数**全ての組合せの和を取れ

```
let is = vec![Some(1i64), None, Some(3)];
let js = vec![Left(4i64), Right(5), Right(6)];
```

```
is.into_iter().flatten()
  .flat_map(|i| {
    js.iter().cloned()
      .flat_map(move |j| {
        j.right()
          .and_then(move |j|
            (j % 2 == 0)
            .then_some(i + j + 100)
    )})}).collect::<Vec<_>>()
```

例3：Iterator の合成

- 例： **is 内の Someの中身** と、**js内のRightの偶数**全ての組合せの和を取れ

```
let is = vec![Some(1i64), None, Some(3)];
let js = vec![Left(4i64), Right(5), Right(6)];
```

```
is.into_iter().flatten()
  .flat_map(|i| {
    js.iter().cloned()
      .flat_map(move |j| {
        j.right()
          .and_then(move |j|
            (j % 2 == 0)
            .then_some(i + j + 100)
    )})}).collect::<Vec<_>>()
```

```
qdo! {Iter {
    Some(i) ← is;
    Right(j) ← js.clone();
    guard j % 2 == 0;
    let k = 100i64;
    return i + j + k
  }}
  .collect::<Vec<_>>()
```

例3：Iterator の合成

- 例： **is 内の Someの中身** と、**js内のRightの偶数**全ての組合せの和を取れ

```
let is = vec![Some(1i64), None, Some(3)];
let js = vec![Left(4i64), Right(5), Right(6)];
```

```
is.into_iter().flatten()
  .flat_map(|i| {
    js.iter().cloned()
      .flat_map(move |j| {
        j.right()
          .and_then(move |j|
            (j % 2 == 0)
            .then_some(i + j + 100)
    )})}).collect::<Vec<_>>()
```

```
qdo! {Iter {
    Some(i) ← is;
    Right(j) ← js.clone();
    guard j % 2 == 0;
    let k = 100i64;
    return i + j + k
  }}
  .collect::<Vec<_>>()
```

こっちの方が
圧倒的可読性💪

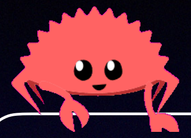
※可読性が高くなる人工的な例を選んでいるといえばそうですが、コードを書いてくれればこの種のロジックはでてきますよね？

😄なるほど、 便利そう！

（と思ってください）

余談

余談：Rust と Haskell のマクロ比較



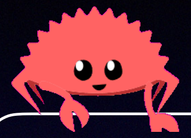
Rust: declarative / proc macro

- declarative macro = macro_rules! のやつ
- **proc macro = トークン列をトークン列に変換**する関数
  - 中間出力は**構文的に不正でもよい**
  - 構文をシームレスに拡張しやすい
    - トークンは共通なのである程度フォーマッタとも相性がいい
  - derive 記法も実体は proc macro
- **実際**：syn クレートでトークン列をマップ
  1. 独自の情報をトークン列からパーズ
  2. 中間形式に変換
  3. 中間形式にトークン列への変換を実装
  - 結構快適
- **難点**：マクロの**修飾対象以外の型・traitの定義情報が取れない**ので全てを**定義サイトに集約**する必要

Haskell: Template Haskell

- **構文木生成**：式・定義・パターン・型
  - 環境に問い合わせて AST を生成
    - **他の型**・クラス・関数の**情報を取得可能**
    - 直接構文木を書く代わりに**クォート**を使って [| $x + $x |] のように書ける
  - 式クォートには**型が付いている版**もある
- **準クォート**を使って独自の構文も定義可能
- **難点**：
  - 全てがクォートで書ける訳ではなく、**構文木は GHC のメジャーバージョンごとに変わる**ので互換性が大変（compat libraryはある）
  - 準クォートは事実上全ての文字列が書けるので、**フォーマッタと相性がわるい**

余談：Rust と Haskell のマクロ比較



Rust: declarative / proc macro

- declarative macro = macro_rules! のやつ
- **proc macro = トークン列をトークン列に変換**する関数
  - 中間出力は**構文的に不正でもよい**
  - 構文をシームレスに拡張しやすい
    - トークンは共通なのである程度フォーマッタとも相性がいい
  - derive 記法も実体は proc macro
- **実際**：syn クレートでトークン列をマップ
  1. 独自の情報をトークン列からパーズ
  2. 中間形式に変換
  3. 中間形式にトークン列への変換を実装
  - 結構快適
- **難点**：マクロの**修飾対象以外の型・traitの定義情報が取れない**ので全てを**定義サイトに集約**する必要

Haskell: Template Haskell

- **構文木生成**：式・定義・パターン・型
  - 環境に問い合わせて AST を生成
    - **他の型**・クラス・関数の**情報を取得可能**
    - 直接構文木を書く代わりに**クォート**を使って [| $x + $x |] のように書ける
  - 式クォートには**型が付いている版**もある
- **準クォート**を使って独自の構文も定義可能
- **難点**：
  - 全てがクォートで書ける訳ではなく、**構文木は GHC のメジャーバージョンごとに変わる**ので互換性が大変（compat libraryはある）
  - 準クォートは事実上全ての文字列が書けるので、**フォーマッタと相性がわるい**

閑話休題

qualified_do まとめ

- **qualified_do**: proptest, Vec, Option, Result 等を統一的に扱える
  - Fallible な計算については、? 記法と相補的に使える
    - ? と違い、**異なる型同士の計算をネスト**できる
    - より直感的なパターンマッチ・条件分岐
  - **実は**：まだクロージャの move との兼ね合いに改善の余地あり
- Rust の proc macro は結構面白い
  - Template Haskell みたいに他モジュールの型定義を reify できたらもっと嬉しいなあ

おしながき

1. **`qualified_do`**: Rust でも (Qualified)do する件について

おしながき

2. do 記法でわかるかもしれない！モナド

do 記法でわかるかもしれない！モナド

😄do記法はべんりそう！

🤔でもなんで

そんなもん作ったの？

A. 毎回

```
i.into_iter().map().zip()
        .enumerate().try_fold()
        .and_then()
```

とかって書くのつらくないですか？😭

そこでモナド！

🤔モナドとは？

副作用の制御に使われるモナドの複雑さのため、Haskellには抵抗を持つ人々がいる。
ワドラーは批判を和らげるために、こう語っている。

**「モナドは単なる自己関手の圏における
モノイド対象だよ。何か問題でも？」**

——James Iry, 不完全にしておよそ正しくないプログラミング言語小史

※ワドラーはそんなこと言ってません
※ワドラーはそんなこと言ってません

圏論的には
そうなんですが

※いいたい事がある人はもうちょっと我慢してね

それはさておき

🤔モナドとは？

A. モナド = do式が使える構造

モナド = do式が使える構造

- モナドの圏論的な定義は「自己函手の圏におけるモノイド対象」
  - これを噛み砕くと右の「Haskellでのモナドの定義」になる
  - といってもやっぱりよくわからない
- それが理論計算機科学で嬉しいのは、**do式が使える**から

🤔どういうことか？

定義（モナド）

$\langle M, \mathtt{pure}, \rightarrowtail\!\!\!\!\Rightarrow \rangle$ が**モナド**

古くは return と呼ばれていた

$\overset{\text{def}}{\iff}$ M は自己函手、`pure :: a → M a`,

`(>>=) :: (a → M b) → (b → M c) → a → M c`

であって、次を満たすもの：

1. $\mathtt{pure} \rightarrowtail\!\!\!\!\Rightarrow \mathtt{f} = \mathtt{f} = \mathtt{f} \rightarrowtail\!\!\!\!\Rightarrow \mathtt{pure}$
2. $\mathtt{f} \rightarrowtail\!\!\!\!\Rightarrow (\mathtt{g} \rightarrowtail\!\!\!\!\Rightarrow \mathtt{h}) = (\mathtt{f} \rightarrowtail\!\!\!\!\Rightarrow \mathtt{g}) \rightarrowtail\!\!\!\!\Rightarrow \mathtt{h}$

「モナドの定義」の Rust 的意味

- 実は「⟹」は **Rust の `.and_then()` や `.flat_map()`** に相当！
  - 厳密には `mx >>= f = (id ⟹ f) mx :: m a → (a → m b) → m b`
- `pure` は `Ok(...)` とか `Some(...)` など、**値を文脈に入れる**操作
- Rust で `and_then()` や `flat_map()` が連続したら、それは**モナドの合図**！

do 式の変換

- **do記法**は次の規則に従って `.and_then()` を使った式に「**翻訳**」することができる：

```
                do { e } ⇝ e
do {x ← e₁; e₂; ..; eₙ} ⇝ e1 >>= \x → do { e₂; ..; eₙ }
do { let x = e₁; e₂; ..; eₙ} ⇝ let x = e1 in do { e₂; ..; eₙ }
```

do 式の変換

- **do記法**は次の規則に従って .and_then() を使った式に「**翻訳**」することができる：

```
                  do { e } ⇝ e
     do {x ← e₁; e₂; ..; eₙ} ⇝ e1 >>= \x → do { e₂; ..; eₙ }
do { let x = e₁; e₂; ..; eₙ} ⇝ let x = e1 in do { e₂; ..; eₙ }
```

➡ モナドの定義は、この**「翻訳」がそれっぽく振る舞う**保証に必要！

「それっぽく振る舞う」とは？

- **例題**：与えられたリストに含まれる自然数の奇数の約数を列挙せよ
- **do記法**で書くならこう

```
do { Iter {
  i ← is;
  j ← (1..=i).filter(move |j| i % j == 0);
  if j % 2 == 0 { vec![] } else { vec![j] }
}}
```

「それっぽく振る舞う」とは？（続）

- **生のIterator** で書くならこう

```
is.into_iter()
  .flat_map(|i|
    (1..=i)
    .filter(move |j| i % j == 0)
  )
  .flat_map(|j|
    (j % 2 == 1).then_some(j)
  )
```

「それっぽく振る舞う」とは？（続）

- **生のIterator** で書くならこう
- でも**こうも書ける**よね？

```
is.into_iter()
  .flat_map(|i|
    (1..=i)
    .filter(move |j| i % j == 0)
  )
  .flat_map(|j|
    (j % 2 == 1).then_some(j)
  )
```

```
is.into_iter()
  .flat_map(|i|
    (1..=i)
    .filter(move |j| i % j == 0)
    .flat_map(|j|
      (j % 2 == 1).then_some(j)
    )
  )
```

「それっぽく振る舞う」とは？（続）

- **生のIterator** で書くならこう
- でも**こうも書ける**よね？

```
is.into_iter()
  .flat_map(|i|
    (1..=i)
    .filter(move |j| i % j == 0)
  )
  .flat_map(|j|
    (j % 2 == 1).then_some(j)
  )
```

二つめの flat_map は**直列**

```
is.into_iter()
  .flat_map(|i|
    (1..=i)
    .filter(move |j| i % j == 0)
    .flat_map(|j|
      (j % 2 == 1).then_some(j)
    )
  )
```

「それっぽく振る舞う」とは？（続）

- **生のIterator** で書くならこう
- でも**こうも書ける**よね？

```
is.into_iter()
  .flat_map(|i|
    (1..=i)
    .filter(move |j| j % i == 0)
  )
  .flat_map(|j|
    (j % 2 == 1).then_some(j)
  )
```

二つめの flat_map は**直列**

```
is.into_iter()
  .flat_map(|i|
    (1..=i)
    .filter(move |
    .flat_map(|j|
      (j % 2 == 1).then_some(j)
    )
  )
```

二つめの flat_map は**内側に移動**！

「それっぽく振る舞う」とは？（続）

- **生のIterator** で書くならこう
- でも**こうも書ける**よね？

```
is.into_iter()
  .flat_map(|i|
    (1..=i)
    .filter(move |j| j % i == 0)
  )
  .flat_map(|j|
    (j % 2 == 1).then_some(j)
  )
```

二つめの flat_map は**直列**

```
is.into_iter()
  .flat_map(|i|
    (1..=i)
    .filter(move |
    .flat_map(|j|
      (j % 2 == 1).then_some(j)
    )
  )
```

二つめの flat_map は**内側に移動**！

- **この二つが実際同値！**というのがモナド則！（演習問題：ちゃんと考えてみて！）

do 式に戻すと

```
do { Iter {
  i ← is;
  do { Iter {
    (1..=i).filter(move |j| i % j == 0)
   if j % 2 == 0 { vec![] } else { vec![j] }
  }}
}}
```

=

```
do { Iter {
  j ← do { Iter {
    i ← is;
    (1..=i).filter(move |j| i % j == 0)
  }}
  if j % 2 == 0 { vec![] } else { vec![j] }
}}
```

😄なるほど

😄Monad って do 式なんだ！

😄Monad って do 式なんだ！

※1: 厳密には「do式がちゃんと振る舞う奴」であって「do式」そのものではありません

🤔ところで、

Functor とか **Applicative**

っていうのも聞いたんだけど……？

🤔これはどういうのなの？

A.「**制限されたdo**が使える奴等」です

Functor, Applicative, Monad, ...

- **Monad は Applicative の一種、Applicative は Functor の一種**
  - いずれも**制限された do 式が使えるもの**と思える
  - Haskell では **ApplicativeDo 言語拡張**を有効化するとこれらにも do 記法が使える

Functor, Applicative, Monad, ...

- **Monad は Applicative の一種、Applicative は Functor の一種**
  - いずれも**制限された do 式が使えるもの**と思える
  - Haskell では **ApplicativeDo 言語拡張**を有効化するとこれらにも do 記法が使える

Functor で使えるdo

```
do {
  x ← e;
  pure (f x)
}
```

Functor, Applicative, Monad, ...

- **Monad は Applicative の一種、Applicative は Functor の一種**
  - いずれも**制限された do 式が使えるもの**と思える
  - Haskell では **ApplicativeDo 言語拡張**を有効化するとこれらにも do 記法が使える

Functor で使えるdo

束縛は一つだけ

```
do {
  x ← e;
  pure (f x)
}
```

Functor, Applicative, Monad, ...

- **Monad は Applicative の一種、Applicative は Functor の一種**
  - いずれも**制限された do 式が使えるもの**と思える
  - Haskell では **ApplicativeDo 言語拡張**を有効化するとこれらにも do 記法が使える

Functor で使えるdo

```
do {
  x ← e;
  pure (f x)
}
```

束縛は一つだけ

x とdo 外で束縛されている変数のみが現れる項

Functor, Applicative, Monad, ...

- **Monad は Applicative の一種、Applicative は Functor の一種**
  - いずれも**制限された do 式が使えるもの**と思える
  - Haskell では **ApplicativeDo 言語拡張**を有効化するとこれらにも do 記法が使える

Functor で使えるdo

```
do {
  x ← e;
  pure (f x)
```

束縛は一つだけ

x とdo 外で束縛されている変数のみが現れる項

Rust の **.map()** 相当！

Functor, Applicative, Monad, ...

- **Monad は Applicative の一種、Applicative は Functor の一種**
  - いずれも**制限された do 式が使えるもの**と思える
  - Haskell では **ApplicativeDo 言語拡張**を有効化するとこれらにも do 記法が使える

Functor で使えるdo

```
do {
  x ← e;
  pure (f x)
```

束縛は一つだけ

x とdo 外で束縛されている変数のみが現れる項

Rust の **.map()** 相当！

Applicative で使えるdo

```
do {
  x1 ← e1;
  ...;
  xn ← en;
  pure (f x1 ... xn)
}
```

Functor, Applicative, Monad, ...

- **Monad は Applicative の一種、Applicative は Functor の一種**
  - いずれも**制限された do 式が使えるもの**と思える
  - Haskell では **ApplicativeDo 言語拡張**を有効化するとこれらにも do 記法が使える

Functor で使えるdo

```
do {
  x ← e;
  pure (f x)
```

束縛は一つだけ

x とdo 外で束縛されている変数のみが現れる項

Rust の **.map()** 相当！

Applicative で使えるdo

```
do {
  x1 ← e1;
  ...;
  xn ← en;
  pure (f x1 ... xn)
}
```

但し **e_j は以前の x_i を含まない**任意の式

Functor, Applicative, Monad, ...

- **Monad は Applicative の一種、Applicative は Functor の一種**
  - いずれも**制限された do 式が使えるもの**と思える
  - Haskell では **ApplicativeDo 言語拡張**を有効化するとこれらにも do 記法が使える

Functor で使えるdo

```
do {
  x ← e;
  pure (f x)
}
```

束縛は一つだけ

x とdo 外で束縛されている変数のみが現れる項

Rust の **.map()** 相当！

Applicative で使えるdo

```
do {
  x1 ← e1;
  ...;
  xn ← en;
  pure (f x1 ... xn)
}
```

但し **e_j は以前の x_i を含まない**任意の式

x_i とdo 外で束縛されている変数のみが現れる項

Functor, Applicative, Monad, ...

- **Monad は Applicative の一種、Applicative は Functor の一種**
  - いずれも**制限された do 式が使えるもの**と思える
  - Haskell では **ApplicativeDo 言語拡張**を有効化するとこれらにも do 記法が使える

Functor で使えるdo

```
do {
  x ← e;
  pure (f x)
```

束縛は一つだけ

x とdo 外で束縛されている変数のみが現れる項

Rust の **.map()** 相当！

Applicative で使えるdo

```
do {
  x1 ← e1;
  ...;
  xn ← en;
  pure (f x1 ... xn)
```

但し **e_j は以前の x_i を含まない**任意の式

x_i とdo 外で束縛されている変数のみが現れる項

Rust の **.zip()** 相当！

Applicative は並列処理可能

- Applicative の do は、「**独立した部分計算同士を最後に zip する**」ようなもの
  - つまり、**個々の束縛の間に依存関係がない**計算
- 依存関係がないので、**Applicativeなdoは各枝を並列化**できることがわかる！
  - 逆に、**無制限の do プログラム**は束縛された**部分計算間に依存関係**があり、条件分岐などが発生する可能性がある！
- ビルドシステムと思うと、**動的な実行グラフを許すのがモナド、実行前に静的に決まるのがApplicative**
  - Shake Build System 論文 [Shake]が面白いです

```
qdo! { BoxedProptest {
  l ← inner.clone();
  r ← inner.clone();
  returnAdd(l, r)
}},
```

||

```
inner.clone()
  .prop_zip(inner.clone())
  .prop_map(|(l,r)| Add(l, r))
```

Applicative は並列処理可能

- Applicative の do は、「**独立した部分計算同士を最後に zip する**」ようなもの
  - つまり、**個々の束縛の間に依存関係がない**計算
- 依存関係がないので、**Applicativeなdoは各枝を並列化**できることがわかる！
  - 逆に、**無制限の do プログラム**は束縛された**部分計算間に依存関係**があり、条件分岐などが発生する可能性がある！
- ビルドシステムと思うと、**動的な実行グラフを許すのがモナド、実行前に静的に決まるのがApplicative**
  - Shake Build System 論文 [Shake]が面白いです

```
qdo! { BoxedProptest {
  l ← inner.clone();
  r ← inner.clone();
  returnAdd(l, r)
}},
```

||

```
inner.clone()
  .prop_zip(inner.clone())
  .prop_map(|(l,r)| Add(l, r))
```

実は proptest は Applicative Do に落ちる例でした

`qualified_do` の実際の脱糖処理

- 最初から ApplicativeDo 相当の処理をしている
  - マクロ側で変換前に**束縛間の依存関係を分析**し、**依存関係がなければ zip と map だけを組み合わせて翻訳する**モードに切り替え
  - パターンマッチなどは **`Alternative`** という「**失敗し得る `Applicative`**」を使って翻訳している
- **演習問題**：
  1. ApplicativeDo だけを考えた翻訳を考えてみよう
  2. Functor (map) や Applicative (zip) にも満たすべき性質がある。Monad則から導き出してみよう。

😄なるほど〜

🤔でも、わざわざ部分で考える意味あるの？

A. Monad ではないが
Applicative な例がある

ZipIter はモナドではない

- これまでみた **Iter の qdo!** は各束縛の候補を**総当たり**していた
  - フィルターして枝刈りを許そうと思うと、自然とこうなる
  - Iterの Applicative Do は、**zip() ではなく product()** 相当の処理になる！
- **ApplicativeDo では Iter に対して zip() を使う**ような実装も考えられる
  - しかし **zip() と両立するような and_then() は実装できない**（演習問題！）
  - **ヒント**：固定長の配列 [T; N] については **zip()** と両立するモナドがある
- qualified_do では qdo! の後に**実装のタグを与える**ので、両方に対して (Applicative) **do を併存させ意味論を切り替える**ことができる！

例：Vec の do 意味論を切り替える

```
let is = vec![1i64, 2]; let js = vec![40i64, 50, 60];
```

```
let ans =  qdo! { UndetVec {
  i ← is;
  j ← js;
  return i + j
}};
assert_eq!(
  ans,
  vec![41, 42, 51, 52, 61, 62]
);
```

```
let ans = qdo! { ZipVec {
  i ← is;
  j ← js;
  return i + j
}};
assert_eq!(
  ans,
  vec![41, 52]
);
```

😄 qualified_do で意味論を

色々差し替えられるんだね！

🤔ところで、Rust でモナドや
Functor はどう定式化するの？

発展的話題：GATs によるRank1多相

- Haskell では **Rank-1 多相**といって、**「型を引数に取る引数」に対する型クラス**が定義できる
- Rust では**全てのジェネリック引数を完全に渡した型**についてしか trait を定義できない
  - そこ**でモナドそのものではなく**、**「実装のタグ」**の方を trait にすることを考える
  - `Iter`, `UndetVec`, `BoxedProptest` 等はこの**実装タグ**
  - Rustでは**ジェネリック引数を取る関連型（Generic Associated Types; GATs）**が定義できる
- `qdo!` は**タグに紐付けられた `map()` や `zip()`** などで翻訳
  - cf. Haskell の `QualifiedDo` では、モジュール名 `M` をつけて `M.do { .. }` などとかく

Haskell

```haskell
class Functor f where
  fmap :: (a → b) → f a → f b
```

Rust

```rust
pub trait Functor {
  type C<T>;

  fn fmap<T, U, F>(f: F, fa: Self::C<T>)
      → Self::C<U>
  where
    F: FnMut(T) → U;
}
```

発展的話題：GATs によるRank1多相

- Haskell では **Rank-1 多相**といって、**「型を引数に取る引数」に対する型クラス**が定義できる
- Rust では**全てのジェネリック引数を完全に渡した型**についてしか trait を定義できない
  - そこ**でモナドそのものではなく**、**「実装のタグ」**の方を trait にすることを考える
  - `Iter`, `UndetVec`, `BoxedProptest` 等はこの**実装タグ**
  - Rustでは**ジェネリック引数を取る関連型（Generic Associated Types; GATs）**が定義できる
- `qdo!` は**タグに紐付けられた `map()` や `zip()`** などで翻訳
  - cf. Haskell の `QualifiedDo` では、モジュール名 `M` をつけて `M.do { .. }` などとかく

Haskell

```haskell
class Functor f where
  fmap :: (a → b) → f a → f b
```

Rust

```rust
pub trait Functor {
  type C<T>;

  fn fmap<T, U, F>(f: F, fa: Self::C<T>)
      → Self::C<U>
  where
    F: FnMut(T) → U;
}
```

GATs を使った定式化の利点

- Haskell では**一つの型に対して一つのインスタンス**のみ実装できる
- Vec vs ZipVec のような対立がある場合、**newtype** を使って**「一般的でない実装」は newtype に対して与える**形になる
  - newtype はだいぶ扱いやすくなったが、それでも**他のモナドの内側だったりすると取り回しが悪く**、つらいところがある
- qualified_do では**実装タグ**を定式化に使っているので**使用感に差はない**
  - ただし、生で呼ぼうとすると実装タグのトレイトメソッドを qualify して呼び出すことになるので重い
  - どうせ do でしか使わないのであまり気にしなくていい

```rust
enum UndetVec {};
impl Functor for UndetVec {
    type C<T> = Vec<T>;
    fn fmap<T, U, F>(f: F, v: Vec<T>) →Vec<U>
    where
        F: FnMut(T) → U,
    {
        v.into_iter().map(f).collect()
    }
}

enum ZipVec {};
impl Functor for ZipVec {
    type C<T> = Vec<T>;
    fn fmap<T, U, F>(f: F, v: Vec<T>) →Vec<U>
    where
        F: FnMut(T) → U,
    {
        v.into_iter().map(f).collect()
    }
}
```

qualified_do と GATs トリックの関係

- `qualified_do` は `qdo! { MyImpl { ... } }` を脱糖する際に `MyImpl::map`, `MyImpl::and_then` などを呼ぶだけ
  - `MyImpl` が先程の **GATs Functor を実装していて、それがスコープにあればそれを使う**
    - GATs トリックを使ったファンクタは `functo_rs` というクレートで提供しています
  - 型制約などから `Functor` にはならないような型は、`Functor` を経由せず**直接こういった関数を実装**すれば、そっちが呼ばれる

do記法とモナド まとめ

- `and_then()` や `flat_map()` が**連鎖**しているプログラムは **do で書き直せる**
- 複数の `and_then` / `flat_map` が絡むと、**一つのdo式に対応するプログラムが複数**現れる
  - **それが全部同値**！というのがモナドの要請
- **`Functor` や `Applicative`** は限定的な do 記法が使える一般化！
  - Functor は厳密には型が成す圏の**自己函手**（Endofunctor）のことです
  - Applicative は**並列化可能性**と関連し、Applicative だが Monad にならないものもある
  - qualified_do では複数の実装を並立させられる
- (Stable) Rust で Functor や Monad を実装する場合、**「実装のタグ」に対してトレイト**を定式化し、**GATs で実際のコンテナを与える**形にすればいける

おしながき

2. do記法でわかるかもしれない！モナド

おしながき

3. Rust 世界の二つのモナド ～資源は大切に～

Rust 世界の二つのモナド

～資源は大切に～

😄モナドって便利なんだ！

😄明日から Rust でも使おう！

✋でもちょっと待って

そのモナド、
どっちのモナドですか？

? ? ?

実は Rust や Linear Haskell
の世界では

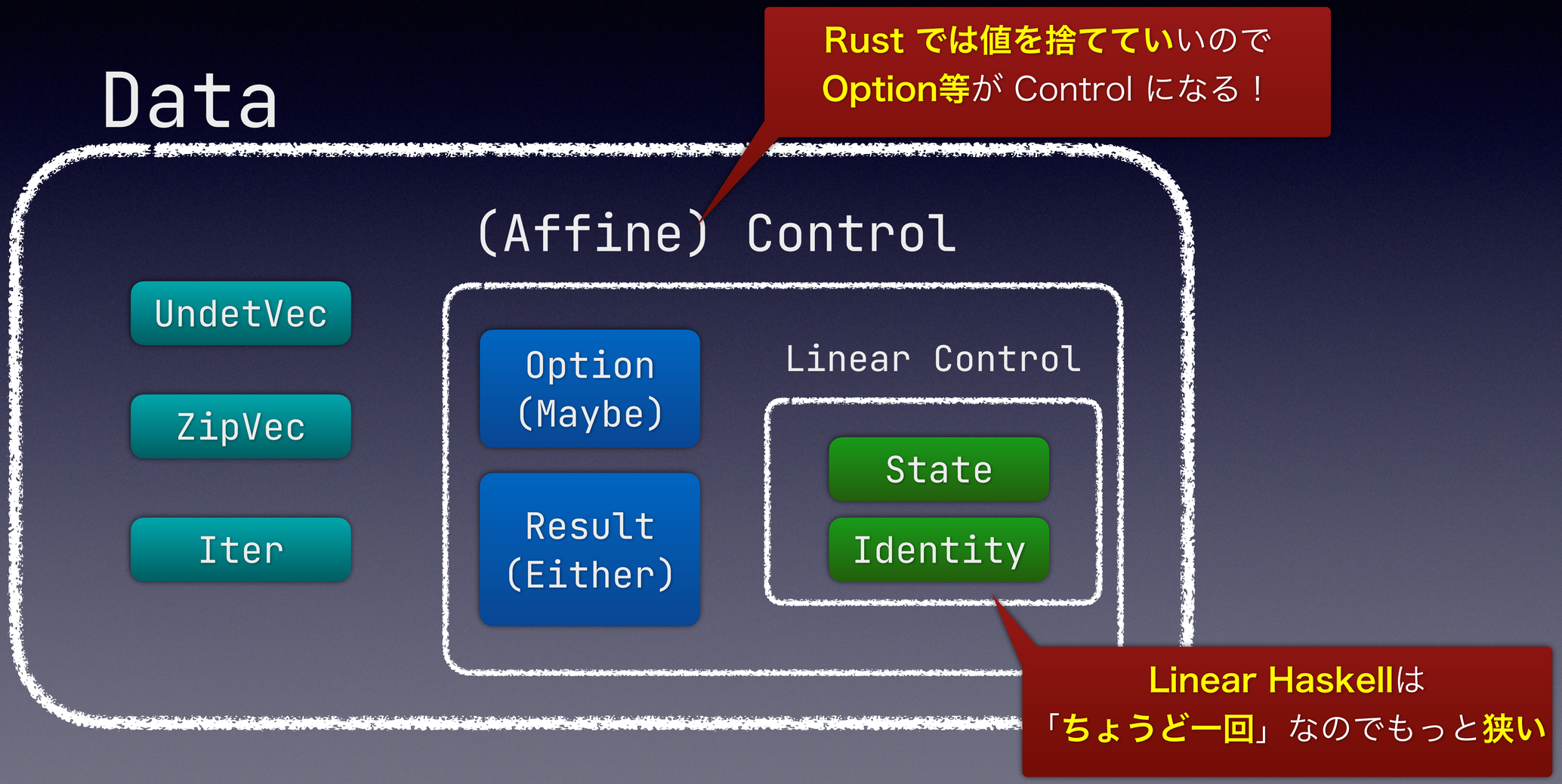
Monad の階層は

二つに分裂します

分裂モナド君劇場

- 渡された関数を**何回使える**かで **Functor の種類が分裂**する
  - Rust は所有権ベース**アファイン型**、Linear Haskell は**線型型**をサポートしており、リソースの利用に厳しい
- **Data Functor**: 値は（高々）一回、**クロージャは0回以上何回でも**呼べる、**「データ型っぽく振る舞う」** Functor の階層
  - Rust でいえば **`FnMut()` を取る**、ということ
- **Control Functor**: 値も**クロージャも一回しか**呼べない、**「制御構造っぽく振る舞う」** Functor の階層
  - Rust でいえば **`FnOnce()` を取る**、ということ
- **Control Functor なら Data Functor** だが、**逆は真ではない**
- Haskell の場合はこの二つに加えて（Linearity を無視した）**「普通の」モナド**の階層が加わる
  - これまでの議論は概ね「普通の」モナドの議論をしていた

Control と Data Functor の例



😮そんな違いがあるんだ！

🤔でもどういう違いが……？

A. 無条件で do が使えるか否か

😰え、モナドなら do ができるんじゃ……？

Recall:

副作用の制御に使われるモナドの複雑さのため、Haskellには抵抗を持つ人々がいる。
ワドラーは批判を和らげるために、こう語っている。

**「モナドは単なる自己関手の圏における
モノイド対象だよ。何か問題でも？」**

——James Iry, 不完全にしておよそ正しくないプログラミング言語小史

※ワドラーはそんなこと言ってません
※ワドラーはそんなこと言ってません

「モナドは単なる自己関手の圏におけるモノイド対象だよ。何か問題でも？」

圏論的には、そう

実は、プログラミングで嬉しいのは

単なるモナドではなく**強モナド**

本当に欲しいのは**強モナド**

- **モナド**：所定の条件を満たす自己函手（Functor）やら自然変換やらがあればいい
  - 函手やら自然変換やらは、**圏を外から眺めて**定義される「関数的な**対応関係**」
    - **圏自体がその対応を知っているとは限らない**！
  - プログラミングでは関数として呼び出したいから、**プログラムを書いている世界の圏が知っていて**ほしい
    - そんなモナドを**自己豊穣モナド**という
- 普通の Haskell の関数の圏 **Hask** や、線型関数の圏 **LHask** は**モノイド閉圏**と呼ばれる特殊な圏で、**自己豊穣モナド ＝ 強モナド**（一般の圏では一致しない）

Data Functor で do はできない

- LHask や Rust の圏で do 式を使うには、**Functor たちが強モナド**でなくてはならない
  - これはつまり、渡される**クロージャも（高々）一回消費するものに対してしか do が使えない**、ということ！

🤔あれ、でも Data Functor にしかならない `UndetVec` や `ZipVec` にも do を使ってなかった？

お詫びと訂正

二つのモナドといったな

実はモナドが三つある〜

第三のモナド：非線型モナド

- Control vs Data Functor の対立があるのは、**リソースに敏感な世界**の話
- これとは別に、**ふつうの Haskell の世界**や、**`Clone` な Rust の型だけを考えた世界のモナド＝非線型モナド**もある
  - 別名**富豪モナド**（わたしが勝手にそう呼んでいる）
  - Rustなら束縛された値を富豪的にドンドン **`clone()`** していく実装
  - `QualifiedDo` 言語拡張は、この**非線型モナドと Control モナドを併用するために提案**された
- `ZipVec` や `UndetVec` はこの非線型モナド（`functo_rs` の `nonlinear`）を実装しているので、**`clone()` によるオーバーヘッドと引き換え**に do れる！
- Clone できる型は沢山あっても一部なので、二つに数えなかった

二つのモナド：まとめ

- 生 Rust やLinear Haskell のような**リソースに厳しい世界**には**二つモナド階層**がある
  - **Control Functor**: クロージャを（高々）一回しか使わず、do 式が使える
  - **Data Functor**: クロージャを何回でも使い得、do式は使えない
- これとは別に、何でもクローンする**非線型モナド階層**（富豪モナド）もいる
  - `Vec` などに `qualified_do` が使えているのはこれのおかげ（だからコストに気を付けてね）
- プログラミングにおける Functor は endofunctor だし、そもそも単なる functor / monad ではなく**強自己函手・強モナド**
  - 「**函手の対応や `>>=` がプログラムとして書ける**」という当然の要請
  - 圏論プロパーの中には「言葉遣いが雑」と怒る尤もな人もいますが、まあプログラミングの文脈なんだから仮定しないと話にならないしこれくらい暗黙の前提として略したっていいだろと僕は思います

おしながき

3. Rust 世界の二つのモナド ～資源は大切に～

おしながき

1. **`qualified_do`**: Rust でも (Qualified)do する件について
2. do記法でわかるかもしれない！モナド
3. Rust 世界の二つのモナド 〜資源は大切に〜

まとめ

まとめ

- **モナド = 無制限のdo記法を使える構造**
  - `Functor = do { x ← mx ; pure (f x) }` 型のdo記法が使える
  - `Applicative = do { x1 ← mx1 ; ...; xn ← mxn ; pure (f x1 ... xn)}` 型のdo記法 (mxi: 互いに独立)
  - Rust でも Generalised Associated Types (GATs) を使って「実装」にタグづけするトリックで実装可能
- Rust や Linear Haskell など線型・アファイン型のある世界では **Monad の階層が二つに分裂**する
  - **Data Functor**: 渡された関数を**複数回**呼び出し得る（**clone なしで do は使えない**）
  - **Control Functor**: 渡された関数は**（Rust ならば高々）一回しか**使われない（**無条件に do が使える**）
    - `Maybe (Option)` は**Linear Haskellでは Control Monad にならない**が、Rust では **(Affine) Control Monad** に！
- **QualifiedDo 記法**：複数の**「モナドっぽい」**クラスに do記法を使えるようにする Haskell の言語拡張
  - `qualified_do: QualifiedDo` + ApplicativeDo + α を Rust で使うための proc macro `qdo!` を提供

参考文献

- [LinHask] J.-P. Bernardy, Boespflug, Newton, S. P. Jones, and Spiwack: *Retrofitting Linear Types*.
  - Linear Haskell の元論文。仕掛け人の Arnaud の Tweag 記事もかなり参考になる。
- [Shake] N. Mitchel: *Shake before Building* / A. Mokhov, N. Mitchel, and S.P. Jones: Build Systems a la carte.
  - Haskell 製ビルドシステム EDSL Shake の論文群。Applicative や Monad などの階層の直感を得るのに結構いい（第2部の元ネタ）
- A. Spiwack: *A Tale of Two Functors or: How I Learned to Stop Worrying and Love Data and Control*.
  - 第3部の元ネタ。Control / Data Functor の概念の初出です。
- GHC User's Guide: Applicative do-notation; Qualified do-notation.
  - GHC の do まわりの言語拡張のマニュアル。
- 今回の発表は私の Zenn の記事を基に一部取り出して膨らませたものです
  - *Haskell* は *Rust* になれるのか？——*2023*年の *Linear Haskell* 体験記 （今の職場にいるのはこの記事を見付けてもらったお陰でもあります）
  - *Haskeller* の異常な愛情：または、生粋の *Haskeller* は転職して *Rust* を一ヶ月半書いて何を思うようになったか
  - *Rust* で柔軟な *do* 式とモナドを使うために作った幾つかのもの

謝辞

- Kory_3 が「この話を関数型まつりでするべき」と薦めてくれました
  - まとめてみて初めて自分の中で整理できた所もあり、感謝🙏
- 今回他の候補として出していた残り二つの話題は、またの機会に！
  1. Linear Haskell の現在と未来——線型型で Haskell のリソース管理はどう変わるか
  2. 走れ Haskell（ブラウザとCloudflare Workersで）：GHC の WASM バックエンドと JS FFI の現在

One more thing...

Jijブース出展中！

ブースでお待ちしています

CEO 山城、CTO 西村、
そして石井がいます！

↓ 目印はJijのロゴ ↓

Jij

We are hiring!

Rustエンジニア など積極採用中！

制度も充実
・フルリモート可能
・エンジニア勉強会
・研究費 / 書籍購入/ 個人予算 補助

御清聴

ありがとう

ございました

Any Questions?

- **モナド = 無制限のdo記法を使える構造**
  - `Functor = do { x ← mx ; pure (f x) }` 型のdo記法が使える
  - `Applicative = do { x1 ← mx1 ; ...; xn ← mxn ; pure (f x1 ... xn)}` 型のdo記法 (mxi: 互いに独立)
  - Rust でも Generalised Associated Types (GATs) を使って「実装」にタグづけするトリックで実装可能
- Rust や Linear Haskell など線型・アファイン型のある世界では **Monad の階層が二つに分裂**する
  - **Data Functor**: 渡された関数を**複数回**呼び出し得る（**clone なしで do は使えない**）
  - **Control Functor**: 渡された関数は**（Rust ならば高々）一回しか**使われない（**無条件に do が使える**）
    - `Maybe (Option)` は**Linear Haskellでは Control Monad にならない**が、Rust では **(Affine) Control Monad** に！
- **QualifiedDo 記法**：複数の**「モナドっぽい」**クラスに do記法を使えるようにする Haskell の言語拡張
  - `qualified_do: QualifiedDo` + ApplicativeDo + α を Rust で使うための proc macro `qdo!` を提供