

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



自己紹介

- ・ 石井 大海 / konn (Twitter: @mr_konn)
- ・ 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！
- ・ 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 .map() や .flat_map() で頑張る

用例1：proptest の Arbitrary

```
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()))
                ),
        ]
    })
}
```

用例1：proptest の Arbitrary

```
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())),
        ],
    ])
}
```



```
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つしか使えない
 - ・ 例 : Option と Result を混ぜるには **クロージャを作つて即座に消費 (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つしか使えない
 - ・ 例 : Option と Result を混ぜるには **クロージャを作つて即座に消費 (IIFE)** する必要
 - ・ 「いまだ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 + 100);
};
```

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つしか使えない
 - ・ 例 : Option と Result を混ぜるにはクロージャを作つて即座に消費 (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 + 100);
};
```

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つしか使えない
 - ・ 例 : Option と Result を混ぜるにはクロージャを作つて即座に消費 (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！
- ・ パター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つしか使えない
 - ・ 例 : Option と Result を混ぜるにはクロージャを作つて即座に消費 (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！
 - ・ パター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 ?
適当な条件と……
引数が Go の時だけ成功
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

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
    }}
};
```

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

複数の型が
共存！※1

局所的に Option

※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

複数の型が
共存！※1

適当な条件と……

引数が Go の時だけ成功

局所的に Option

※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

複数の型が共存！※1

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！

※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

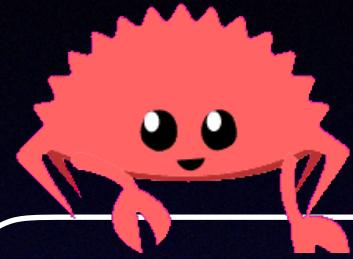


なるほど、便利そう！

(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

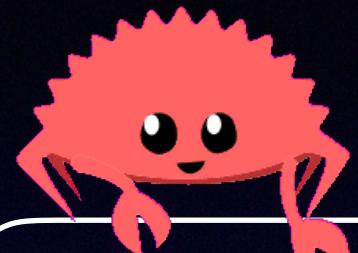
- declarative macro = macro_rules! のやつ
- **proc macro** = トークン列をトークン列に変換する関数
 - 中間出力は構文的に不正でもよい
 - 構文をシームレスに拡張しやすい
 - トークンは共通なのである程度フォーマッタとも相性がいい
 - derive 記法も実体は proc macro
- **実際** : syn クレートでトークン列をマップ
 1. 独自の情報をトークン列からパーズ
 2. 中間形式に変換
 3. 中間形式にトークン列への変換を実装
- 結構快適
- **難点** : マクロの修飾対象以外の型・traitの定義情報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 **クオート** を使って `[|$x + $x|]` のように書ける
- 式クオートには型が付いている版もある
- **準クオート**を使って独自の構文も定義可能
- **難点** :
 - 全てがクオ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
 - 互換性が大変 (compat libraryはある)
 - 準クオ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 **クオート** を使って `[|$x + $x|]` のように書ける
- 式クオートには型が付いている版もある
- **準クオート**を使って独自の構文も定義可能
- **難点** :
 - 全てがクオ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
 - 互換性が大変 (compat libraryはある)
 - 準クオ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
 - ・ 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. 每回

```
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！

do記法とモナドまとめ

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～

二つのモナド：まとめ

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

- ・ **Control Functor**: 渡された関数は (Rust ならば高々) 一回しか使われない

- ・ 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

- Linear Haskell の論文
- Shake before Building の論文
- Tale of Two Functors の記事
- ApplicativeDo や QualifiedDo のリファレンス
- わたしの Zenn

御清聴

ありがとうございます

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

- ・ **Control Functor**: 渡された関数は (Rust ならば高々) 一回しか使われない

- ・ 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