わいわいswiftc #28

Publisher@resultBuilder

Twitter @iceman5499

resultBuilderの簡単な説明

Swift5.1から登場(※)した、式を宣言する形式の文法でコードを記述できるようにする仕組み。

https://github.com/apple/swift-evolution/blob/main/proposals/0289-result-builders.md

```
// SwiftUIの例
@ViewBuilder var body: some View {
  VStack {
    Text("Hello")
    Spacer()
  }
}
```

`@ViewBuilder` はresultBuilderの一種で、SwiftUI.frameworkによって用意されている。`@ViewBuilder` 自体は言語機能ではなくあくまで実装。自分たちで同じものを作ることが出来る。

※Swift5.1時点では@_functionBuilderという名前だった

今回の目的

Combineフレームワークを用いる際の手間のかかる`Publisher`の組み立てをresultBuilderを使って楽できるようにする。

Combineで感じる課題: Publisherの分岐が大変

```
func f() -> AnyPublisher<Int, Error> {
  nanikaPublisher
    .flatMap { v -> AnyPublisher<Int, Error> in // 返り値の型は省略できないことが多い
      if v.isXxx {
        return PassthroughSubject<Int, Never>()
          .setFailureType(to: Error.self) // Never川をError川として返す場合はエラー型を指定
          .eraseToAnyPublisher() // 型消去がほぼ必須
      } else {
        return PassthroughSubject<Int, Error>()
          .eraseToAnyPublisher() // 型消去が分岐ごとに必要
      }
    }
    .eraseToAnyPublisher()
}
```

flatMapを書く際など、型の変換が面倒だったり型推論がうまくいかなくて困ることがある。

解決方法

resultBuilderで型推論器にヒントを与え、かつ間に処理を差し込んで暗黙的な変換ができるようにする

```
func f() -> AnyPublisher<Int, Error> {
  nanikaPublisher
    .flatMapBuild { v in
      if v.isXxx {
        PassthroughSubject<Int, Never>()
      } else {
        PassthroughSubject<Int, Error>()
      }
    }
    .eraseToAnyPublisher()
}
```

resultBuilderを使うと↑をvalidなコードとすることができる。

今回の流れ

1. `eraseToAnyPublisher()` を省略できるようにする
2. 型情報を伝搬させる
3. エラー型を集約できるようにする
   3-2. エラー型を集約できるようにする2
4. 返り値から型パラを与えられるようにする

1. `eraseToAnyPublisher()` を省略できるようにする

1. `eraseToAnyPublisher()` を省略できるようにする

```
.flatMap { v -> AnyPublisher<[String], Never> in
  if v == 0 {
    return Just([])
      .eraseToAnyPublisher() // 👈
  } else {
    return PassthroughSubject<[String], Never>()
      .eraseToAnyPublisher() // 👈
  }
}
```

Combineの川やオペレータはRxSwiftやReactiveSwiftと違いそれぞれが専用の型を持つ。分岐が発生すると複数種類の型を返す必要が生まれ、そのために毎回型消去が必要になる。

→ `buildEither` を使って2つの型を1つにまとめる。

1. `eraseToAnyPublisher()` を省略できるようにする

```
enum EitherPublisher<L: Publisher, R: Publisher>: Publisher
where
  L.Output == R.Output, L.Failure == R.Failure
{
  typealias Output = L.Output
  typealias Failure = L.Failure
  case left(L)
  case right(R)

  func receive<S>(subscriber: S) where S : Subscriber, Failure == S.Failure, Output == S.Input {
    switch self {
    case .left(let value):
      value.receive(subscriber: subscriber)
    case .right(let value):
      value.receive(subscriber: subscriber)
    }
  }
}
```

2つの川の型情報を持てる型を用意する。

1. `eraseToAnyPublisher()` を省略できるようにする

```
@resultBuilder
struct PublisherBuilder {
  static func buildBlock<C: Publisher>(_ component: C) -> C {
    component
  }

  static func buildEither<F: Publisher, S: Publisher>(first component: F) -> EitherPublisher<F, S>
  where
    F.Output == S.Output,
    F.Failure == S.Failure
  {
    .left(component)
  }

  static func buildEither<F: Publisher, S: Publisher>(second component: S) -> EitherPublisher<F, S>
  where
    F.Output == S.Output,
    F.Failure == S.Failure
  {
    .right(component)
  }
}
```

`@resultBuilder` の `buildEither` を使って型をラップする。

1. `eraseToAnyPublisher()` を省略できるようにする

最後に簡単なextensionを用意する。

```
extension Publisher {
  func flatMapBuild<O, P>(
    @PublisherBuilder _ builder: @escaping (Output) -> P
  ) -> Publishers.FlatMap<P, Self>
  where O == P.Output, P: Publisher, P.Failure == Failure
  {
    flatMap(builder)
  }
}
```

1. `eraseToAnyPublisher()` を省略できるようにする

以下のように書けるようになった🙌

```
.flatMapBuild { v in // この{からresultBuilderのスコープが展開している
  if v == 0 {
    Just<[String]>([])
  } else {
    PassthroughSubject<[String], Never>()
  }
}
```

`EitherPublisher<Just<[String]>, PassthroughSubject<[String], Never>>` 型が最終的なPublisherの型になる。

2. 型情報を伝播させる

2. 型情報を伝播させる

次のコードは現状ではビルドできない。

```
let _: AnyPublisher<[String], Never> = PassthroughSubject<Int, Never>()
  .flatMapBuild { v in
    Just([])
  }
  .eraseToAnyPublisher() // Type of expression is ambiguous without more context
```

2. 型情報を伝播させる

`buildExpression` を使う

> buildExpression(_ expression: Expression) -> Component is used to lift the results of expression-statements into the Component internal currency type. It is optional, but when provided it allows a result builder to distinguish Expression types from Component types or to provide contextual type information for statement-expressions.

https://github.com/apple/swift-evolution/blob/main/proposals/0289-result-builders.md#result-building-methods

2. 型情報を伝播させる

```
@resultBuilder
struct PublisherBuilder<P: Publisher> {
  static func buildExpression(_ expression: P) -> P {
    expression
  }
  ...
```

これだけで問題のコードがビルドできるようになる。

```
let _: AnyPublisher<[String], Never> = PassthroughSubject<Int, Never>()
  .flatMapBuild { v in // ② P.Output==Output==[String]だと伝搬する
    Just([]) // ③ exprがPublisher where Output==[String]であると伝わる
  }
  .eraseToAnyPublisher() // ① 返り値からOutput==[String]だと推論される
```

2. 型情報を伝播させる

実はまだ足りない。以下のコードはビルドできない。

```
let _: AnyPublisher<[String], Never> = PassthroughSubject<Int, Never>()
  .flatMapBuild { v in
    if v == 0 {
      Just([])
    } else {
      Empty<[String], Never>() // Cannot convert value of type ↩
      // 'Empty<[String], Never>' to expected argument type 'Just<[String]>'
    }
  }
  .eraseToAnyPublisher()
```

`buildExpression` によって `PublisehrBuilder<P>` の `P` が `Just<[String]>` であると確定してしまうため、それ以外の式を記述できない。

2. 型情報を伝播させる

解決策？（失敗例）

```
struct PublisherBuilder<P: Publisher> {
  static func buildExpression<E>(_ expression: E) -> E {
    expression
  } // オーバーロードを追加
  ...
```

🤔任意の式も受け取れるようにすればいいのではないか？

2. 型情報を伝播させる

解決策？（失敗例）

```
.flatMapBuild { v in // Cannot convert value of type ↩
// 'EitherPublisher<Just<[String]>, Empty<[String], Never>>' to closure result type 'Just<[String]>'
  if v == 0 {
    Just([])
  } else {
    Empty<[String], Never>()
  }
}
```

11ページの `flatMapBuild` の定義で `builder` は `P` を返すことになっているので、それ以外の型が組み上がったとしてもエラーになってしまう。

2. 型情報を伝播させる

解決策（成功例）

`PublisherBuilder` の型パラを `Output` と `Failure` に分割する。

```
@resultBuilder              // ↓変更
struct PublisherBuilder<Output, Failure: Error> {
  ...
}
```

```
extension Publisher {
  func flatMapBuild<O, P>(
    @PublisherBuilder<O, Failure> _ builder: @escaping (Output) -> P
  ) -> Publishers.FlatMap<P, Self>
  where O == P.Output, P: Publisher, P.Failure == Failure
  {
    flatMap(builder)
  }
}
```

https://github.com/sidepelican/PublisherBuilder/blob/waiwai/Sources/WaiwaiPublisherBuilder2-2/main.swift

3. エラー型を集約できるようにする

3. エラー型を集約できるようにする

次のコードは分岐中のエラーの型が一致していないため、コンパイルできない。いい感じにエラー型を変換してくれると嬉しい。

```
let _: AnyPublisher<[String], Error> = PassthroughSubject<Int, Never>()
  .flatMapBuild { v in
    if Bool.random() {
      PassthroughSubject<[String], Never>() // 分岐の中で
    } else {
      PassthroughSubject<[String], Error>() // 異なるエラー型を持つ
    }
  }
  .eraseToAnyPublisher()
```

2つ問題点がある。

- `flatMapBuild` が上流と下流の間のエラー型の変換を許していない
- `PublisherBuilder` の `buildEither` で `F.Failire == S.Failure` としているので分岐の中で異なるエラー型を持つ川を返せない

`flatMapBuild` がエラー型の変換を許していない

これは簡単で、オーバーロードを増やすだけでいい（Combineの `flatMap` も同様に複数のオーバーロードがある）。

```
extension Publisher {
  func flatMapBuild<O, P>(
    @PublisherBuilder<O, Failure> _ builder: @escaping (Output) -> P
  ) -> Publishers.FlatMap<P, Self>
  where O == P.Output, P: Publisher, P.Failure == Failure
  {
    flatMap(builder)
  }

  func flatMapBuild<O, P>(
    @PublisherBuilder<O, Never> _ builder: @escaping (Output) -> P
  ) -> Publishers.FlatMap<Publishers.SetFailureType<P, Failure>, Self>
  where O == P.Output, P: Publisher, P.Failure == Never
  {
    if #available(macOS 11.0, iOS 14.0, *) {
      return flatMap(builder)
    } else {
      return flatMap { builder($0).setFailureType(to: Failure.self) }
    }
  }
}

extension Publisher where Failure == Never {
  func flatMapBuild<O, F, P>(
    @PublisherBuilder<O, F> _ builder: @escaping (Output) -> P
  ) -> Publishers.FlatMap<P, Publishers.SetFailureType<Self, P.Failure>>
  where O == P.Output, F == P.Failure, P: Publisher
  {
    if #available(macOS 11.0, iOS 14.0, *) {
      return flatMap(builder)
    } else {
      return setFailureType(to: P.Failure.self).flatMap(builder)
    }
  }

  func flatMapBuild<O, P>(
    @PublisherBuilder<O, Never> _ builder: @escaping (Output) -> P
  ) -> Publishers.FlatMap<P, Self>
  where O == P.Output, P: Publisher, P.Failure == Never
  {
    flatMap(builder)
  }
}
```

3. エラー型を集約できるようにする

`buildEither` で `F.Failire == S.Failure` としているので分岐の中で異なるエラー型を持つ川を返せない

`F.Failire == S.Failure` が成り立つように中の式の型を変形してあげればいい。
外側の型から内側の型に触れるためには・・？ → `buildExpression`

3. エラー型を集約できるようにする

`buildEither` で `F.Failire == S.Failure` としているので分岐の中で異なるエラー型を持つ川を返せない

```
static func buildExpression<P: Publisher>(_ expression: P) -> Publishers.SetFailureType<P, Failure>
where
  P.Output == Output, P.Failure == Never
{
  expression.setFailureType(to: Failure.self)
}
```

これを追加するとコンパイルできるようになる。`buildEither` に到達する前に一度このエラー型変換 `buildExpression` を通過することでエラー型が合致する。

3-2. エラー型を集約できるようにする2

さらに、次のコードもコンパイルできるようになってほしい。

```
let _: AnyPublisher<[String], Error> = PassthroughSubject<Int, Never>()
  .flatMapBuild { v in
    if Bool.random() {
      PassthroughSubject<[String], MyError>() // Static method ↩
// 'buildExpression' requires the types 'MyError' and 'Never' be equivalent
    } else {
      PassthroughSubject<[String], Error>()
    }
  }
  .eraseToAnyPublisher()
```

3-2. エラー型を集約できるようにする2

同様に `buildExpression` を追加して内部で `mapError` するだけで解決するはず・・・？

```
static func buildExpression<C: Publisher>(_ component: C) -> Publishers.MapError<C, Failure>
where
  C.Output == Output, Failure == Error
{
  component.mapError { $0 as Error }
}
```

3-2. エラー型を集約できるようにする2

単に新しい `buildExpression` を追加すると、エラー型がNeverのときにエラーになる。

```
.flatMapBuild { v in
  if Bool.random() {
    PassthroughSubject<[String], Never>() // Ambiguous use of 'buildExpression'
  } else {
    PassthroughSubject<[String], Error>()
  }
}
```

Candidates:

- `func buildExpression<P: Publisher>(_ expression: P) -> P`
- `func buildExpression<C: Publisher>(_ component: C) -> Publishers.MapError<C, Failure> where where C.Output == Output, Failure == Error`

3-2. エラー型を集約できるようにする2

オーバーロード優先度を調節することで`Ambiguous use of`エラーを解決できる。

詳しくはわいわいswiftc #16を参照。

```
@_disfavoredOverload // つけた
static func buildExpression<C: Publisher>(_ component: C) -> Publishers.MapError<C, Failure>
where C.Output == Output, Failure == Error {
  component.mapError { $0 as Error }
}

@_disfavoredOverload // つけた
static func buildExpression<P: Publisher>(_ expression: P) -> Publishers.SetFailureType<P, Failure>
where P.Output == Output, P.Failure == Never {
  expression.setFailureType(to: Failure.self)
}

// より狭い型の条件を持つ関数を追加
@_disfavoredOverload
static func buildExpression<P: Publisher>(_ expression: P) -> Publishers.SetFailureType<P, Failure>
where P.Output == Output, P.Failure == Never, Failure == Error {
  expression.setFailureType(to: Failure.self)
}
```

Swiftのオーバーロード選択のスコア規則12種類 https://gist.github.com/omochi/587ce6e7089c763007b46ce8fa3f27a4

4. 返り値から型パラを与えられるようにする

`PublisherBuilder` が作る型は複雑になりやすい

→ クロージャの返り値の型をあえて明示的に指定することが難しい

```
let aaa = PublisherBuilder.build { // @PublisherBuilderのスコープを展開するだけの関数
  if Bool.random() {
    PassthroughSubject<Int, CustomError>()
      .map { _ in "" }
  } else {
    PassthroughSubject<String, Never>()
  }
}
```

`aaa` は `EitherPublisher<Publishers.Map<PassthroughSubject<Int, CustomError>, String>, Publishers.SetFailureType<PassthroughSubject<String, Never>, CustomError>>`

4. 返り値から型パラを与えられるようにする

`buildFinalResult` で最終的な返り値にパターンを与えられる。

```
static func buildFinalResult<C: Publisher>(_ component: C) -> C {
  component
}

@_disfavoredOverload
static func buildFinalResult<C: Publisher>(_ component: C) -> AnyPublisher<Output, Failure>
where
  Output == C.Output, Failure == C.Failure
{
  component.eraseToAnyPublisher()
}
```

としておくと、`{ v -> AnyPublisher<String, Never> in` のような形で型を指定できるようになる。

ここでもオーバーロードの優先順位を落とすことで、何も指定しなければ型消去をしない元々の型を返すようになっている。

4. 返り値から型パラを与えられるようにする

返り値から型のヒントを与えることで中の式の推論を強化できる。

```
.flatMapBuild { v -> AnyPublisher<[String], Never> in
  if Bool.random() {
    Just([])
  } else {
    Empty()
  }
}
```

下の例はどちらも有効。

```
let _: PassthroughSubject<Int, Error> = PublisherBuilder.build {
    PassthroughSubject<Int, Error>()
}
let _: AnyPublisher<Int, Error> = PublisherBuilder.build {
    PassthroughSubject<Int, Error>()
}
```

その他利用場面

テストでモックに差し込むときに便利だった

```
class MockHoge: HogeProtocol {
  var fooHandler: ((Int) -> AnyPublisher<Int, Error>)?
  func foo(_ v: Int) -> AnyPublisher<Int, Error> { fooHandler!(v) }
}

let mock = MockHoge()
mock.fooHandler = build { _ in // @PublisherBuilderのスコープを展開する関数
  Just(1)
}
```

おわり

その他のケースにも対応した最終的な `PublisherBuilder` はこちらに公開されています。

https://github.com/sidepelican/PublisherBuilder