Traversableを理解する

塚田竜勇孫

ある日APIでこのようなコード書いていた

```
let getRecipientIds
  (fetchCompanyListIds: CompanyId -> Async<CompanyListId>)
  (fetchDashboardAuthors: CompanyListId -> Async<DashboardAuthor>)
  (companyIds: CompanyId list)
  : Async<RecipientIds> =
  async {
      let! companyListIds =
        companyIds // CompanyId list
        |> List.map fetchCompanyListIds  // Async<CompanyListId> list
         (*ここでList<Async<CompanyListIds>>をAsync<List<CompanyListId>>にしたい*)

      let! dashboardAuthors =
        companyListIds
        |> List.map fetchDashboardAuthors
        (*ここでList<Async<DashboardAuthor>>をAsync<List<DashboardAuthor>>にしたい*)

      return RecipientIds(dashboardAuthors |> List.map (fun (DashboardAuthor x) -> RecipientId x))
  }
```

結論こう書ける

```
open FSharpPlus

let getRecipientIds
  (fetchCompanyListId: CompanyId -> Async<CompanyListId>)
  (fetchDashboardAuthor: CompanyListId -> Async<DashboardAuthor>)
  (companyIds: CompanyId list)
  : Async<RecipientIds> =
  async {
    let! companyListIds = traverse fetchCompanyListId companyIds
    let! dashboardAuthors = traverse fetchDashboardAuthor companyListIds
    return RecipientIds(dashboardAuthors |> List.map (fun (DashboardAuthor x) -> RecipientId x))
  }
```

なぜこう書けるのか

Traversable

Data structures that can be traversed from left to right, performing an action on each element.

Minimal complete definition

- `traverse f x` | `sequence x`

```
static member Traverse (t: 'Traversable<'T>, f: 'T -> 'Applicative<'U>) : 'Applicative<'Traversable<'U>>
static member Sequence (t: 'Traversable<'Applicative<'T>>) : 'Applicative<'Traversable<'T>>
```

Other operations

- `gather f x` | `transpose x` (same as `traverse` and `sequence` but operating on ZipApplicatives)

```
static member Gather (t: 'Traversable<'T>, f: 'T -> 'ZipApplicative<'U>) : 'ZipApplicative<'Traversable<'U>>
static member Transpose (t: 'Traversable<'ZipApplicative<'T>>) : 'ZipApplicative<'Traversable<'T>>
```

Rules

```
t << traverse f = traverse (t << f)
traverse Identity = Identity
traverse (Compose << map g << f) = Compose << map (traverse g) << traverse f
```

Related Abstractions

- Functor: A traversable is generic on the Traversable type parameter and the (Applicative) Functor inner type parameter.
- Applicative: An applicative is a functor whose `map` operation can be splitted in `return` and `(<*>)` operations.
- Foldable : All traversables are foldables.

順を追って理解する

Functor

「ある構造の中のそれぞれの要素に関数を適用する」という考え方の抽象

map関数を提供するデータ構造のこと。

FSharpPlusだとこれらのFunctorに汎用的に使えるmap関数が使えたりする

定義

```
Map: (x: 'Functor<'T>, f: 'T -> 'U) : 'Functor<'U>
```

Functorの例

`seq<'T>`

`list<'T>`

`array<'T>`

`Async<'T>`

`Result<'T,'U>`

`Task<'T>`

Applicative

純粋な式を埋め込んだり、一連の計算を行い、その結果を結合する（<*>）操作を提供するFunctor
「ある構造の中のそれぞれの要素に関数を適用するという考え方」の抽象

定義

```
Return (x: 'T) : 'Applicative<'T>
```

```
(<*>) (f: 'Applicative<'T -> 'U>, x: 'Applicative<'T>) : 'Applicative<'U>
```

その他の関数

```
Lift2 (f: 'T1 -> 'T2 -> 'T, x1: 'Applicative<'T1>, x2: 'Applicative<'T2>)
: 'Applicative<'T>
```

Applicativeの例

`seq<'T>` `Result<'T, 'U>`
`list<'T>` `Task<'T>`
`array<'T>` `Async<'T>`

Foldable

要約された値に畳み込むことができるデータ構造

最小の定義

```
ToSeq (x:'Foldable<'T>) :seq<'T>
```

その他の関数

```
FoldMap (x:'Foldable<'T>, f:'T->'Monoid)
```

Foldableの例

```
seq<'T>
```

```
list<'T>
```

```
option<'T>
```

```
voption<'T>
```

Traversable

コレクション内の各要素にアクションを実行しながら、左から右へ走査(traverse)できるデータ構造。

最小の定義

```
Traverse (t: 'Traversable<'T>, f: 'T -> 'Applicative<'U>) :
'Applicative<'Traversable<'U>>
```

```
Sequence (t: 'Traversable<'Applicative<'T>>) :
'Applicative<'Traversable<'T>>
```

Traversableの例

- `seq<'T>`
- `list<'T>`
- `array<'T>`
- `option<'T>`
- `Result<'T, 'Error>`

実際に作って確認してみる

まずList<Async<'a>>をAsync<List<'a>>に変化するinvertという関数を実装してみることにする。

```
let getRecipientIds
    (fetchCompanyListId: CompanyId -> Async<CompanyListId>)
    (fetchDashboardAuthor: CompanyListId -> Async<DashboardAuthor>)
    (companyIds: CompanyId list)
    : Async<RecipientIds> =
    let rec invert (x: List<Async<'a>>): Async<List<'a>> =
        match x with
        | [] -> failwith "TODO"
        | head::tail -> failwith "TODO"
    async {
        let! companyListIds =
          companyIds
          |> List.map fetchCompanyListId
          |> invert

        let! dashboardAuthors =
          companyListIds
          |> List.map fetchDashboardAuthor
          |> invert
        return RecipientIds(dashboardAuthors |> List.map (fun (DashboardAuthor x) -> RecipientId x))
    }
```

まずパターンマッチした結果、[]だった場合、Asyncで包まれた空のリストを返す

```
let rec invert (x: List<Async<'a>>) =
  match x with
  | [] -> async.Return []
  | head::tail -> failwith "TODO"
```

それ以外の場合は、headと再起呼び出ししたinvert tailの結果をリストにして返している

```
let rec invert (x: List<Async<'a>>): Async<List<'a>> =
  match x with
  | [] -> async.Return []
  | head::tail -> Async.map2 (fun h t -> h::t) head (invert tail)
```

野生のApplicativeを発見した

パターンマッチに注目してほしい。map2やReturnはasyncをアプリカティブたらしめるものにしている。

Applicativeの定義

```
Return (x: 'T) : 'Applicative<'T>
```

```
(<*>) (f: 'Applicative<'T -> 'U>, x: 'Applicative<'T>) : 'Applicative<'U>
```

```
let rec invert (x: List<Async<'a>>): Async<List<'a>> =
  match x with
  | [] -> async.Return []
  | head::tail -> Async.map2 (fun h t -> h::t) head (invert tail)
```

Asyncはアプリカティブなのでapplicativeに定義されているlift2という関数を使うこともできる

```
let rec invert (x: List<Async<'a>>): Async<List<'a>> =
  match x with
  | [] -> async.Return []
  | head::tail -> lift2 (fun h t -> h::t) head (invert tail)
```

Applicativeの<*>とresultを使った形でも書くことができる

```
let rec invert (x: List<Async<'a>>): Async<List<'a>> =
  match x with
  | [] -> result []
  | head::tail -> result (fun h t -> h::t) <*> head <*> (invert tail)
```

最終的なコードは以下

invert関数を使うことによって、`List<Async<'a>>`を`Async<List<'a>>`に変換することができた

```
let getRecipientIds
  (fetchCompanyListId: CompanyId -> Async<CompanyListId>)
  (fetchDashboardAuthor: CompanyListId -> Async<DashboardAuthor>)
  (companyIds: CompanyId list)
  : Async<RecipientIds> =
  let rec invert (x: List<Async<'a>>): Async<List<'a>> =
      match x with
      | [] -> async.Return []
      | head::tail -> Async.map2 (fun h t -> h::t) head (invert tail)
  async {
      let! companyListIds =
        companyIds
        |> List.map fetchCompanyListId
        |> invert

      let! dashboardAuthors =
        companyListIds
        |> List.map fetchDashboardAuthor
        |> invert
      return RecipientIds(dashboardAuthors |> List.map (fun (DashboardAuthor (x: string)) -> RecipientId x))
  }
```

野生のSequenceを発見した

定義したinvertはsequenceとして知られているTranversable型に定義されている関数

sequenceは何かしらのラップされた値のコレクションを受け取り、何かしらのラップされたコレクションに変える。
FsharpPlusにすでにsequenceが定義されているので、これを使うとこのようになる。

```
open FSharpPlus

let getRecipientIds
  (fetchCompanyListId: CompanyId -> Async<CompanyListId>)
  (fetchDashboardAuthor: CompanyListId -> Async<DashboardAuthor>)
  (companyIds: CompanyId list)
  : Async<RecipientIds> =
  async {
    let! companyListIds =
      companyIds
      |> List.map fetchCompanyListId
      |> sequence

    let! dashboardAuthors =
      companyListIds
      |> List.map fetchDashboardAuthor
      |> sequence
    return RecipientIds(dashboardAuthors |> List.map (fun (DashboardAuthor x) -> RecipientId x))
  }
```

もっと良くなる余地がある

companyIdsやcompanyListIdsなどのリストを複数回処理をしている

```
let getRecipientIds
  (fetchCompanyListId: CompanyId -> Async<CompanyListId>)
  (fetchDashboardAuthor: CompanyListId -> Async<DashboardAuthor>)
  (companyIds: CompanyId list)
  : Async<RecipientIds> =
  let rec sequence (x: List<Async<'a>>): Async<List<'a>> =
      match x with
      | [] -> async.Return []
      | head::tail -> Async.map2 (fun h t -> h::t) head (invert tail)
  async {
      let! companyListIds =
        companyIds
        |> List.map fetchCompanyListId
        |> sequence

      let! dashboardAuthors =
        companyListIds
        |> List.map fetchDashboardAuthor
        |> sequence
      return RecipientIds(dashboardAuthors |> List.map (fun (DashboardAuthor (x: string)) -> RecipientId x))
  }
```

sequenceに関数を渡せるようにしてみる

```
let rec sequence f x =
  match x with
  | [] -> async.Return []
  | head::tail -> Async.map2 (fun h t -> h::t) (f head) (sequence f tail)
```

```
let getRecipientIds
  (fetchCompanyListId: CompanyId -> Async<CompanyListId>)
  (fetchDashboardAuthor: CompanyListId -> Async<DashboardAuthor>)
  (companyIds: CompanyId list)
  : Async<RecipientIds> =
  let rec sequence f  x =
    match x with
    | [] -> async.Return []
    | head::tail -> Async.map2 (fun h t -> h::t) (f head) (sequence f tail)
  async {
    let! companyListIds = sequence fetchCompanyListId companyIds

    let! dashboardAuthors = sequence fetchDashboardAuthor companyListIds

    return RecipientIds(dashboardAuthors |> List.map (fun (DashboardAuthor (x: string)) -> RecipientId x))
  }
```

野生のTraverseを発見した

さきほど書いたこちらの関数はtraverseと呼ばれているTraversableに定義されている関数。

シーケンスとマッピングの両方を同時に行う場合、関数traverseを呼び出すとスッキリ書ける。

fsharp plusにももちろん定義されているので、こう書ける。

```
open FSharpPlus
let getRecipientIds
  (fetchCompanyListId: CompanyId -> Async<CompanyListId>)
  (fetchDashboardAuthor: CompanyListId -> Async<DashboardAuthor>)
  (companyIds: CompanyId list)
  : Async<RecipientIds> =
  async {
    let! companyListIds = traverse fetchCompanyListId companyIds

    let! dashboardAuthors = traverse fetchDashboardAuthor companyListIds

    return RecipientIds(dashboardAuthors |> List.map (fun (DashboardAuthor x) -> RecipientId x))
  }
```

Traversableはsequenceとtraverseという２つの関数を持つ。

sequenceはtraverseの引数fに`id`という同一性関数をカリー化させたもの

```
let sequence = traverse id
```

もう一度traverseの定義を見てみる

```
static member Traverse (t: 'Traversable<'T>, f: 'T -> 'Applicative<'U>) : 'Applicative<'Traversable<'U>>
```

先程見たようにAsyncはApplicative

TraversableになるにはFunctorかつFoldableである必要がある

リストはFunctorでもあり、FoldableでもあるのでTraversable

traverseは `T → Applicative<U>` と `Traversable<T>` を受け取り、データ構造内の値を受け取った関数で変換し、その結果を `Applicative` 内に格納されたデータ構造に集約するもの

まとめ

- sequenceが有用なシーン
  - optionやResult、Asyncのようなものでラップされた値のコレクションを持っていて、実際に必要なのはoption<list<'a>>やResult<list<'a>, 'e>、Async<List<’a>,’e>などである場合
- traverseが有用なシーン
  - ラップされた値のコレクションに対して計算を実行する必要がある場合
  - マッピングとシーケンスを両方実行したい場合

参考

- https://fsprojects.github.io/FSharpPlus/abstraction-traversable.html
- https://dev.to/choc13/grokking-traversable-bla
- https://qiita.com/suin/items/0255f0637921dcdfe83b
- 『Programming Haskell 2nd edition』