プログラミング言語論 #07

Type Constraints

2018-05-28

型変数の導入によりどのような型であっても引数として受け付ける関数を定義することができるようになった.

しかしこれは実際のプログラミングにおいて制約が緩すぎて使いづらいことが多い.

今回は

- 型名を使って唯一の型のみを受け付ける
- 型変数を使って全ての型を受け付ける

の間を埋めるような型指定を導入する.

型変数が必要な引数を持つ関数の利点

一つの関数定義によって複数の型に対応できる.

C 言語には型変数がない

```c
// 要素数を数える
int count_i (int v[]) { .. }

// 要素数を数える
int count_d (double v[]) { .. }

// 二分木を構成する要素数を数える
int count_b (struct Btree *tree) { .. }
```

型変数を持つ Haskell なら. . .

```haskell
-- 要素数を数える
count :: a -> Int
count v = ...
```

１回定義するだけ

問題点

意味がない計算のことを考える必要がある

- 意味がなさそう： Bool の割り算，ウィジェットの割り算
- 定義できなさそう：雲の写真が約数を 2 つ持つかどうか判定する関数
- 意味あるかも：住所データの足し算＝マージ， mp3 の足し算＝連結
- 意味ありそう：リストの長さ，リストのソート

- 関数の引数が特定の１つの型に対応 – 面倒くさい
- 関数の引数が全ての型に対応 – 緩すぎ

確認：関数，演算子の本当の型

足し算に対応する演算子

```
2 + 4
[] + []
True + False
0.2 + 0.4    -- 初出
```

上記の結果を踏まえて (+) の型を答えよ

等価関係に対応する演算子

```
4 == 5
True == False
[] == []
div == mod   -- 初出
```

上記の結果を踏まえて (==) の型を答えよ

型の部分集合を導入する

- 特定の 1 つの型を指定する＝（対象数 1）
- 中間に当たるもの＝型全体の作る集合の（有限）部分集合
- 型変数を用いる＝型全体の作る集合（無限集合）

方法

型 $\in$ 集合 2 層モデル

型を要素とする集合を定義する．ある型は複数の集合に属せる．集合の指定により対象を限定する．集合の集合は考えない．

型階層モデル

型をまとめたものも型とする．まとめた型は要素になっている型のどれでもよいことを意味する．再帰的に繰り返せる．型は複数の型に属してもよい．最適な型を指定して対象を限定する．

Haskell の場合

型 $\in$ 集合 2 層モデルを採用

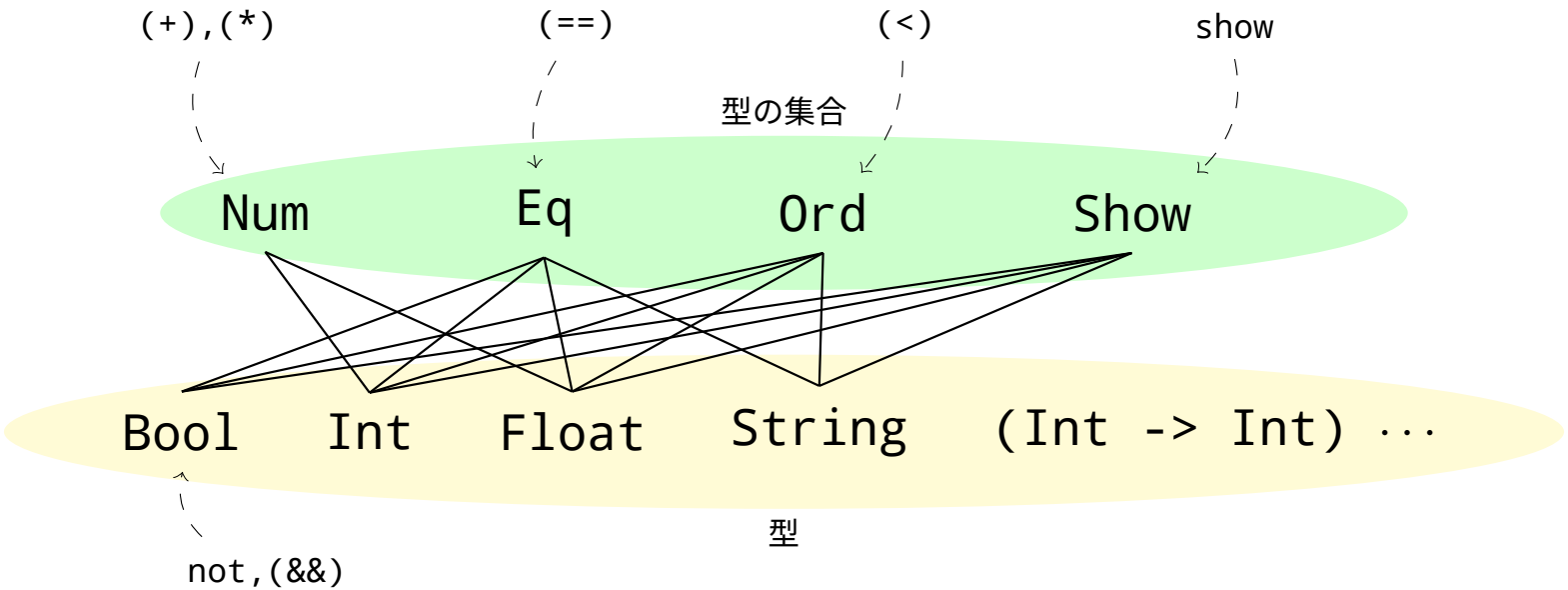
- 既に集合がいくつか定義済み，型は追加済み
- プログラマは自由に，集合の定義，集合への型の追加，ができる
- 意味がない型はその集合に入れなければよい
- 型はいくつの集合に入れられてもよい

Haskell には型も集合も大量に存在（数学的知識の反映，プログラマからの要請）．特に数値関係は非常に多い．

Haskell での定義済み集合の例

※このページの内容は今後の試験の根拠となる

集合名	基準	備考
Num	数 (number) とみなせる型の集合	試験対象
Eq	等しい (equal) かどうか判定可能な型の集合	試験対象
Ord	全順序 (total order) がある型の集合	試験対象
Show	画面に表示できる (show) 型の集合	



プログラマによる拡張の場面例

じゃんけんゲームの作成中に：

- グー + グーができるべきだと思う人はじゃんけんというものを集合 Num に追加
- グー == グー ができるべきだと思う人はじゃんけんというものを集合 Eq に追加
- グー < チョキ ができるべきだと思う人はじゃんけんというものを集合 Ord に追加

goo が 0 で choki は 1 と考えてしまう人には意味がわからない（この人の頭の中にはじゃんけん型は存在しない．あるのは int のみ）

C で D&D

```
1  Wizard a, b;          // 構造体を使ってこう書けたとする
2  if (a < b) ....;      // 必要あるか，便利か，書きたいか
```

Haskell での実演

必要な知識

1. 新しい型を定義する方法
2. 型集合に型を追加する方法
3. 新しい型集合を定義する方法
4. 特定の型集合にのみ使える関数を定義する方法

1. 型の定義

Haskell における新データ型の定義文

```
1  data GCP = Goo | Choki | Par     -- | は「または」を意味
```

これにより

- 型 GCP(左辺)
- 定数 Goo，定数 Choki，定数 Par （右辺）

が定義される

M_PI, True と書く方が 3.1， 1 と書くよりもよいのと同じ理由で定義する

2. 型集合に型を追加する

instance C T ...

型 T を集合 C に追加する．. . . には通常 where ... が続く．

型集合は Haskell では実際には型クラスという．

型クラスへの追加例

```
1  -- Goo + Goo と書きたい
2  instance Num GCP where
3    Goo + Goo = Choki    -- Numの仲間に入るにはせめて+の定義が必要
4    x + y = Goo          -- 2回定義している？→ ▸ 場合分け
5
6  -- Goo == Goo と書きたい
7  instance Eq GCP where
8    Goo == Goo = True    -- Eqの仲間に入るにはせめて==の定義が必要
9    x == y = False       -- 「x == y」が定義したい左辺
```

既に定義済みの型クラスに定義済みの型を追加する場合には実際には where 以下に色々と書かなければいけない

2. 新しい型クラスを定義する

class C a ...

型クラス C を定義する．. . . には通常 where ... が続く．その中で型名 a を使うはず．

型クラス定義例

```
1  -- IntとGCPだけ受け付けるようなことがしたいとする
2  -- そのための型クラスを自分で用意する
3  class IorJ a          -- whereを使わない例
4
5  class C1 a where      -- 使う例
6    f :: a -> a -> Int  -- メソッドのプロトタイプ宣言のようなもの
```

3A. 型クラスに含まれる型全部に対する関数を定義する

関数名 :: C a => これまでの型宣言の構文

型変数 a を使う型宣言である．ただし a が取りうる型は型クラス C に含まれていなければならないという制約が付いた．

「C a =>」を型制約という．

準備： Int と GCP だけを対象にするため型クラスへ追加

```
1 instance IorJ Int
2 instance IorJ GCP
```

IorJ クラスに含まれる型だけ受付ける関数の宣言と定義

```
1 double :: IorJ a => a -> [a]
2 double a = [a,a]
```

3'. 複数の型制約

関数名 :: (C_1 a, C_2 b, ⋯) => ⋯

タプルの形で型制約を記述する

例

```haskell
f :: (Eq a, Ord a) => a -> [a]          -- 1つの型変数に2つ制約
g :: (Eq a, Ord b) => a -> b -> Bool    -- それぞれ制約
h :: (Show a, Eq a, Ord b) => ...       -- 必要なだけ列挙
```

3B. 型クラスに含まれる型ごとに関数を定義する

「2. 型クラスに型を追加する」タイミングで行う

準備：型クラスの修正

```
1  class IoJ a where
2    marui :: a -> Bool    -- クラスメソッドの宣言のみ
```

instance 宣言の修正

```
1  instance IoJ Int where
2    marui 0 = True    -- ここには宣言は書かない．定義のみ
3    marui 8 = True
4    marui x = False
5
6  instance IoJ GCP where
7    marui Goo = True
8    marui x = False
```

:type とエラーメッセージ読解

型推論の結果は聞けば教えてくれる

f を宣言なしで定義

```
1  -- f ::
2  f x y = x < y
```

人間が f を宣言

```
1  f :: a -> a -> Bool
2  f x y = x < y
```

型推論の結果を表示

```
1  > :type f
2  f :: Ord a => a -> a -> Bool
3  > :type (f 8 3) -- なんでも受付け
4  (f 8 3) :: Bool
```

型検査の結果

```
1  <interactive>:7:9: error:
2      • No instance for (Ord a) arising from a use of
           ‘<’
3        Possible fix:
4          add (Ord a) to the context of
5            the type signature for:
6              f :: a -> a -> Bool
```

型推論の結果表示

1. 場所... でエラー
2. (<) を利用しているのに (Ord a) の制約がない
3. 解決案：
4. (Ord a) を追加しなさい

:type とエラーメッセージ読解

型推論の結果は聞けば教えてくれる

f を宣言なしで定義

```
1  -- f ::
2  f x y = x < y
```

人間が f を宣言

```
1  f :: a -> a -> Bool
2  f x y = x < y
```

型推論の結果を表示

```
1  > :type f
2  f :: Ord a => a -> a -> Bool
3  > :type (f 8 3) -- なんでも受付け
4  (f 8 3) :: Bool
```

型検査の結果

```
1  <interactive>:7:9: error:
2      • No instance for (Ord a) arising from a use of
           ‘<’
3        Possible fix:
4          add (Ord a) to the context of
5            the type signature for:
6              f :: a -> a -> Bool
```

型推論の結果表示

(+), (==) を型推論せよ

1. 場所... でエラー
2. (<) を利用しているのに (Ord a) の制約がない
3. 解決案：
4. (Ord a) を追加しなさい

利点再掲

型クラスに含まれる型にだけ使える関数を定義できるようになった．必要なら後から型クラスに型を入れることもできる．

Wikipedia

実装例 ［編集］

ここでは、3層フィードフォワードニューラルネットワークで回帰を実装する。$x = [-1, 1]$ において、$y = 2x^2 - 1$ を学習する。活性化関数は ReLU を使用。学習は確率的勾配降下法（バックプロパゲーション）を使う。

3層フィードフォワードニューラルネットワークのモデルの数式は以下の通り。X が入力、Y が出力、T が訓練データで全て数式では縦ベクトル。ψ は活性化関数。W_1, W_2, B_1, B_2 が学習対象。B_1, B_2 はバイアス項。

$$Y = W_2\psi(W_1X + B_1) + B_2$$

誤差関数は以下の通り。誤差関数は出力と訓練データの間の二乗和誤差を使用。

$$E = \frac{1}{2}\|Y - T\|^2$$

C 言語

```
1  double Y[10], W2[10], W1[10], X[10],
       ...
2
3  Y = W2*(W1 * X + B1) ...; // エラー
```

配列には (*)，(+) が定義されていない．そして定義できない．プログラム大変．

別の言語 1

```
1  W2, X, B1, B2,...
2
3  Y = W2 * (W1 * X + B1) ... ;
```

配列に (*) が定義されていた．コピペするだけでプログラム終了

別の言語 2, 例えば Haskell

```
1  instance Num 配列 where -- 自分で定義
2    a * b = ....
3
4  y = w2*(w1*X ... -- 最初エラー，定義後OK
```

配列には (*) が定義されていなかったので，自分で配列を Num に追加した．後はコピペ

第２回第９シート：型と関数・演算子の本当の対応

数値に関して

Num は ▸ シート 8 で定義

- 加減乗, div, mod :: (Num a) => a -> a -> a

String 型

実は喋ってないことがあるのだが無視できるので省略

Bool 型

等価性に関して

Eq は ▸ シート 8 で定義

- (==), (/=) :: (Eq a) => a -> a -> Bool

順序性に関して

Ord は ▸ シート 8 で定義

- (<), (<=), ... :: (Ord a) => a -> a -> Bool

Haskell に関する補足：パターンマッチング

- 関数定義はパターンマッチングを用いて「複数回定義する」ようなことができる

```
1  f 定数1 = 定義1          -- 実引数が定数1の場合に対応
2  f 定数2 = 定義2          -- 実引数が定数2の場合に対応
3  ...
4  f 仮引数 = 最終定義      -- 実引数がそれ以外の全てに対応
5
6  fa 1 = 1
7  fa 2 = 1
8  fa n = fa (n - 1) + fa (n - 2)
```

途中に別の関数の定義が入ると二重定義エラーになるのでまったく自由に「複数回定義」できるわけではない