プログラミング言語論 #05

Type Variables

2018-05-14

Outline

複数の値を返したい別の状況に対応する型の拡張を学ぶ．その導入の必然的帰結として，型が変数で表される状況が存在することを理解し，型システムの拡張の必要性を理解する．

複数の値を返したい2

タプルでは対応できない状況

```
1  // 奨学金対象者の番号を全員分返す
2  ??? getAllGradeAA (int year) {...
3    時間が掛かる処理
4    return 一人目の番号;    // 一人もいないかも知れない
5    return 二人目の番号;    // 複数人いるかも知れない
6    ...
7  }
```

-	要素数	要素間の関係
基本型	1	-
タプル	任意個 (≥ 2) ただし固定個	任意

複数の値を返したい2

タプルでは対応できない状況

```
1  // 奨学金対象者の番号を全員分返す
2  ??? getAllGradeAA (int year) {...
3    時間が掛かる処理
4    return 一人目の番号;    // 一人もいないかも知れない
5    return 二人目の番号;    // 複数人いるかも知れない
6    ...
7  }
```

-	要素数	要素間の関係
基本型	1	-
タプル	任意個 (≥ 2) ただし固定個	任意
	任意個 ≥ 0	

複数の値を返したい2

タプルでは対応できない状況

```
1  // 奨学金対象者の番号を全員分返す
2  ??? getAllGradeAA (int year) {...
3    時間が掛かる処理
4    return 一人目の番号;    // 一人もいないかも知れない
5    return 二人目の番号;    // 複数人いるかも知れない
6    ...
7  }
```

-	要素数	要素間の関係
基本型	1	-
タプル	任意個 (≥ 2) ただし固定個	任意
	任意個 ≥ 0	任意

複数の値を返したい2

タプルでは対応できない状況

```
1  // 奨学金対象者の番号を全員分返す
2  ??? getAllGradeAA (int year) {...
3    時間が掛かる処理
4    return 一人目の番号;    // 一人もいないかも知れない
5    return 二人目の番号;    // 複数人いるかも知れない
6    ...
7  }
```

-	要素数	要素間の関係
基本型	1	-
タプル	任意個 (≥ 2) ただし固定個	任意
	任意個 ≥ 0	
	任意個 ≥ 0	固定

List

不定個，同一型

```haskell
a ::
a = [1, 2]

b ::
b = [1 + 3, mod 10 3, 0, -8, -1]

c ::
c = [True]

d ::
d = [(1+2,2-1), (3,4), (5,5), (0,0)]

e ::
e = ([1,3,4], [2,3,5,7])
```

※ a ～ e はそれぞれの行で自動的に生成され，プロセスのメモリ消費量が少しずつ増える．生成以外の操作（追加，分割など）でも結果に相当するデータが自動的に生成され，プロセスのメモリ消費量が少しずつ増える．

リストに関するライブラリ関数, 演算子

注意：前述の [,] はリストの記述構文とし，演算子とは考えないことにする

(:)

- 左引数：**要素**
- 右引数：左引数と**同じ型**[1]の要素から構成された**リスト**
- 結果：結合されたリスト

リストとリストを結合する演算子ではない

リストとリストを結合する演算子ではない理由

```
1  1 : [1, 22, 3]              -- 型の合った，要素とリストの結合
2  [1] : [1, 22, 3]            -- 型の合った，リストとリストの結合
3  [1, 22, 3] : 1              -- 型の合った，リストと要素の結合
4  "A" : ["BB", "CC"]          -- 型の合った，要素とリストの結合
5  33 : ["BB", "CC", "DD"]     -- 型の合わない，要素とリストの結合
```

[1] 引数間にこのような制約関係があるという点は (==) などの比較演算子と似ている

利用例

関数の返値として利用

```
1  f1 ::
2  f1 x = [x, x+1, x+2]
3
4  f2 ::
5  f2 x = [x < 0, x == 0, 0 < x]
6
7  f3 ::
8  f3 x = x : [1]
```

関数の引数として利用

```
9  f4 ::
10 f4 x = x == [False, False, True, True]
```

タプルとの組合せ

```
11 f5 ::
12 f5 x y = [(x, x == False), (True, y)]
```

注意：タプルとリストの違い

長さに関する違い

```
1  --| タプル長は2以上
2  (1) == 1                    -- 要素数1のタプルはない（要素そのもの）
3  ()                          -- 要素数0のタプル？ 考えないことにする
4  --| リスト長は0以上
5  1 == [1]                    -- 長さが1のリストと要素は別の型
6  [] == [1]                   -- 長さが0のリストは存在する
```

型に関する違い

```
1  --| タプルは要素数が違うと別の型
2  (1, 3) == (1, 2, 3)
3
4  --| リストは要素の型が同じなら長さが違っても同じ型
5  [1] == [1,2,3]
```

同じ型の要素を任意個まとめるための合成型

言語の違い

C

△ 配列を持つがプログラマが細かく指定

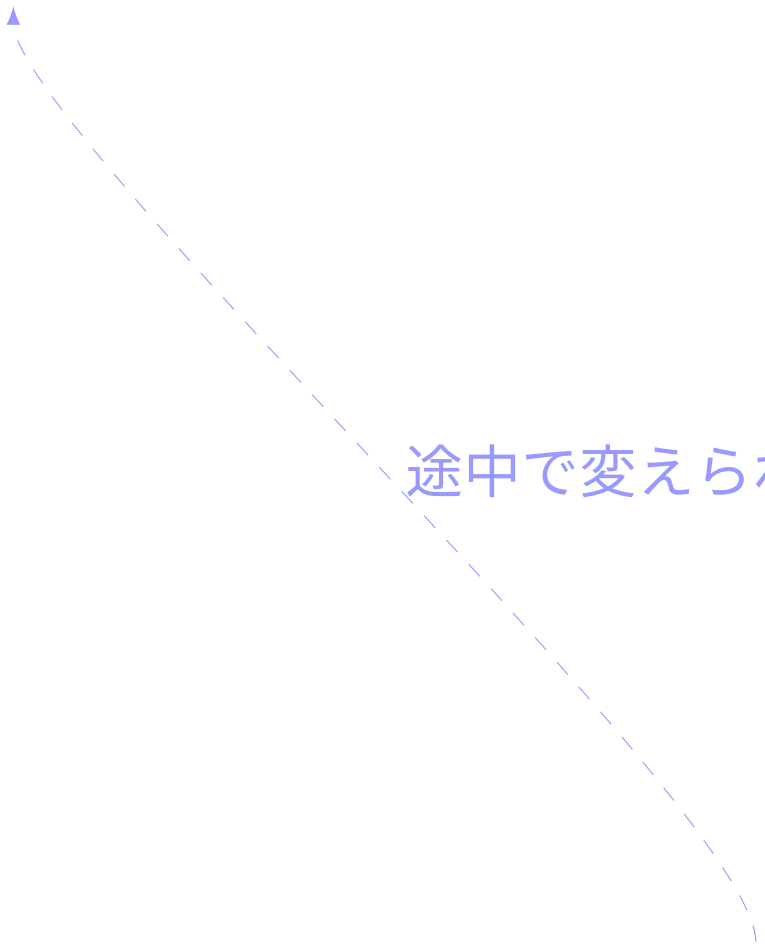
C++, Haskell, Java, JavaScript, Python

○ 少なくとも以下のどちらかを持つ：

- リスト：$O(n)$
- 配列：$O(1)$

ただし，いくつかの言語では型に関する新たな問題が生じる

なお， JavaScript, Python は弱い型付け言語であることから必然的に


途中で変えられる

リストに関する言語固有の特徴的な機能

Python

拡大された添字と辞書

```python
a = ['a','b','c','d']
a[0]
a[1]
a[-1]   # negative index
a[0:3]  # slicing
a[-3:-1]
# dictionary
b = {"ichi":"one", "ni":
     "two"}
b["ichi"]
```

dictionary は配列ではないが関連するので参考として紹介.

Haskell

部分取り出しと途中省略

```haskell
-- いずれも試験に出す予定なし
a :: [Int]
a = [0, 1, 2, 3, 4, 5]
a !! 0      -- 要素アクセス
a !! 1
take 3 a      -- 途中まで
drop 3 a      -- 途中から
[1 .. 10]     -- 有界数上げ
[1, 3 .. 18] -- 等差数列
[1..]         -- 以下省略
```

リストの持つ自由度

本当に定義から型宣言は一意に決まるのか

[ ]

空リストに関する問題

空リストではない x，空リスト y のそれぞれの型を求めよ

```
1  -- 確認1
2  1 : [3]       -- Int と Intのリスト→エラーにならない
3  1 : []        -- Int と ？？？→エラーにならない
4
5  -- 確認2
6  x ::          -- エラーにならないように宣言せよ
7  x = [3]
8  1 : x         -- エラーになってはいけない
9
10 -- 問題
11 y ::          -- エラーにならないように宣言せよ
12 y = []
13 1 : y         -- エラーになってはいけない
```

前問の続き

```
11 True : y
```

数学の復習とプログラミングへの応用

$$f(x) = \ldots x \ldots$$

右辺をよく見ると解が決まる

- $x^2 - 2x + 1 = 0$ $\rightarrow$ 一意
- $x^2 = 1,\ 2x = x + x$ $\rightarrow$ 有限（無限）個
- $x = x + 1$ $\rightarrow$ 不解

定義式（文）の右辺から型が決まる.

定義文から得られる情報が少なければ変数 x の型は

プログラミング言語論専門用語

7

type variable, 型変数

型の変数.

Haskell では小文字で始める． 1 文字である必要はない.

型変数が必要な変数が存在するなら型変数が必要な関数も存在

型に関する方程式を立て，それぞれの型を求めよ

```
1  f ::
2  f x = 3
3
4  g ::
5  g x = True
```

言語の違い

（宣言時に使える）型変数を持つ

C++（auto のこと）, Haskell, Java

型変数を持たない

- JavaScript, Python はそもそも宣言時に型を使用しない
- C は型推論しない，プログラマは 1 つの型を明示しなければならない

例

以下の各関数についてコンパイラの型推論の結果を答えよ

```
1  f ::
2  f x = (x, x)
3
4  g ::
5  g x y = (x, y)
6
7  h ::
8  h x y = [x, y]
9
10 f1::
11 f1 x y = x
12
13 f2 ::
14 f2 x y = 1
```

注意： p.12 で述べたように型推論は定義時の情報しか使わない

```
1  x = []              -- xの定義行：この時点ではよくわからない
2  y = x == [2]        -- xの利用行：右辺よりxの型が[Int]になりそう×
3  -- 定義行 だけで型推論する．定義行でないL.2はxの型推論に参加しない
```

模範解答の一意性について

実は Haskell は型推論をする → 実は型宣言文は書かなくてもよい

書く場合：人間が型を指示，コンパイラが型検査

```
1 f :: Int -> Int
2 f x = 3
```

書かない場合：コンパイラが型推論，コンパイラが型検査

```
1 f x = 3          -- a -> Int ニチガイナイ
```

- コンパイラ：問題なければ人間優先
- 試験の問題文：「コンパイラの型推論の結果を答えよ」

型変数の命名規則のみ（出現順に $a, b, c \cdots$）決めておけば正解は１つ