

パラメトリックな多相性(教3.2節)

プログラミングAIII

2025年度講義資料 (5)

新潟大学 工学部工学科 知能情報システムプログラム

青戸等人

```
# fun selfPair x = (x,x);
val selfPair = fn : ['a. 'a -> 'a * 'a]
```

- selfPairの型は、ある程度決まっていますが、引数の型が'aなら、返り値の型は'a * 'aとなる。つまり型がパラメータ化されている。'aは、型を表わす変数なので、**型変数**とよばれる。
- このように、関数の型が型変数を使って表わされ、変数が具体化されることで、関数がさまざまな型に対応するような多相性を、**パラメトリックな多相性**とよぶ。
- 型変数を使って表わされる型を多相型とよぶが、SMLでは、例外や参照型といった機能と併用するために、多相性をもつ関数定義に制約がある(教3.5節)。

関数の多相性
リスト

目次

① 関数の多相性

② リスト

関数の多相性
リスト

型推論と型チェック(教3.1節)

- SMLでは、関数定義式から最も一般的な型を計算し、コンパイル時に関数の型として返してくれる。
- このように自動的に、(最も一般的な)型を与える仕組みは、**型推論**とよばれる。
- 一方で、式の型に整合性がとれなければ、型エラーを報告してくれる。コンパイル時に型の整合性をチェックする仕組みは、**静的型チェック**とよばれる。
- 静的型チェックや型推論は、モダンなプログラミング言語における重要な仕組み。

```
# fun getLeft (x,y) = x;
val getLeft = fn : ['a, 'b. 'a * 'b -> 'a]
# fun round (x,y,z) = (y,x,z);
val round = fn : ['a, 'b, 'c. 'a * 'b * 'c -> 'b * 'a * 'c]
```

関数の多相性
リスト

目次

① 関数の多相性

② リスト

関数の多相性
リスト

等値演算を使う型(教3.6節)

SMLの型には、等値性判定が使える型(整数など)と等値性判定が使えない型(実数など)があった。以下の関数 isEqual の引数は、等値性判定(=)が使える型という制約が必要。制約は、'a#eqの#eqの部分で表わされている。

```
# fun isEqual (x,y) = if x = y then 1 else 0;
val isEqual = fn : ['a#eq. 'a * 'a -> int]
# isEqual (0,1);
val it = 0 : int
# isEqual ("aa","aa");
val it = 1 : int
# isEqual (0.0, 1.0); (* 実数には利用できない *)
(interactive):9.0-9.17(195) Error:
  (type inference 016) operator and operand don't agree
  operator domain: 'PJB#eq * 'PJB#eq
  operand: 'PIY::{real, real32} * 'PJA::{real, real32}
```

関数の多相性
リスト

多相性

関数の多相性
リスト

明示的な型宣言(教3.3節)

```
# fun selfPair x = (x,x);
val selfPair = fn : ['a. 'a -> 'a * 'a]
# selfPair 1;
val it = (1, 1) : int * int
# selfPair true;
val it = (true, true) : bool * bool
# selfPair "aaa";
val it = ("aaa", "aaa") : string * string
#
```

- ここで定義した関数 selfPair は引数がどのような型であっても計算できる。
- このように、関数などが複数の型に対応できることを**多相性**とよぶ。

- 最も一般的な型で使いたくない場合は、関数に型制約をつけることができる。
- 型制約は変数の後ろに「: 型」という形で書く。
- 以下の例では、以前に定義した getLeft や round に型制約をつけている。型制約を付けなかった場合と得られる型を較べてみよう。

```
# fun getLeft (x,y:int) = x;
val getLeft = fn : ['a. 'a * int -> 'a]
# fun round (x:'t, y:'t, z) = (y,z,x);
val round = fn : ['a, 'b. 'a * 'a * 'b -> 'a * 'b * 'a]
#
```

関数の多重定義(教3.4節)

`+` は, `real` 型に対しても, `int` 型に対しても定義されている. このように, 関数がさまざまな型に対して多重に定義されているようなケースも, 多相性の一種である.

```
# 1 + 1;
val it = 2 : int
# 1.0 + 1.0;
val it = 2.0 : real
# op +;
val it = fn : ['a::{int, ...}. 'a * 'a -> 'a]
#
```

もっとも, このような多相性は多くのプログラミング言語で導入されている.

リストの評価とリストの等価性

```
# 1::2::[];
val it = [1, 2] : int list
# [1, 1+1, 1+1+1]; (* リストの値は, 要素の式の値のリスト *)
val it = [1, 2, 3] : int list
# [1,2,3] = 1::[2,3]; [2,2+1] = [1+1,3];
val it = true : bool
val it = true : bool (* リストの等しさは, その値の等しさ *)
# [1] = [1,1]; [1,2] = [2,1];
val it = false : bool (* 個数も順番も関係ある *)
val it = false : bool
# [1.0] = [1.0]; (* 等価性の判定できない要素のとき *)
(interactive):15.0-15.12(31) Error:
  (type inference 026) operator and operand don't agree
  operator domain: 'PXB#eq * 'PXB#eq
  operand: 'PWJ::{real, real32} list * 'PXA::{real, real
```

8 / 20

12 / 20

関数の多相性
リスト関数の多相性
リスト

目次

リストの基本関数(教6.4節) (1)

① 関数の多相性

② リスト

```
# hd [1,2,3]; tl [1,2,3];
val it = 1 : int
val it = [2, 3] : int list
# null []; null [1];
val it = true : bool
val it = false : bool
# [1,2]@[3];
val it = [1, 2, 3] : int list
# []@[1,2]; ["abc"]@[];
val it = [1, 2] : int list
val it = ["abc"] : string list
# length [1,2]; length [];
val it = 2 : int
val it = 0 : int
# rev [1,2,3];
val it = [3, 2, 1] : int list
```

- `hd`と`tl`はそれぞれ, 先頭要素と先頭要素を取り除いた残りのリストを返す. 空リストに対してはエラー (Empty例外をなげる).
- `null`は空リストかどうかを判定する関数.
- 中置演算`@`は2つのリストを繋げる. 空リストを繋げても同じリストのまま.
- `length`はリストの長さを返す関数.
- `rev`はリストを反転したリストを返す関数

9 / 20

13 / 20

関数の多相性
リスト関数の多相性
リスト

リスト(教6.1節, 教6.2節)

リストの基本関数(教6.4節) (2)

組型に続き, もっとも基本的な複合型の1つであるリスト型について学習する.

リスト

- リストとは, リスト構造によって実現されている**同じ型の要素の列**のことである. 要素が型`'a`をもつとき, そのリストの型は`'a list`となる.
- 列 a_1, \dots, a_n のリストを $[a_1, \dots, a_n]$ と書く.
- 空リストを`[]`と書く. (`nil`ともよぶ.)
- リスト`xs`の先頭に要素`x`をつけたリストを, `x::xs`と書く.
- 中置演算子`::`(`cons`ともよぶ)は右結合で, リスト $[a_1, \dots, a_n]$ は, $a_1::\dots::a_n::[]$ の略記である.

```
# concat ["aa", "bb", "cc"];
val it = "aabbcc" : string
# implode;
val it = fn : char list -> string
# implode ["#a", "#b", "#c"];
val it = "abc" : string
# explode;
val it = fn : string -> char list
# explode "hello";
val it = ["#h", "#e", "#l", "#l", "#o"] : char list
#
```

- `concat`は文字列のリストを受けとり, リストの文字列を連結して得られる文字列を返す関数
- `implode`は文字のリストを受けとり, それらを繋げて得られる文字列を返す.
- `explode`は, 文字列を受けとり, その文字列を分解した文字のリストを返す.

10 / 20

14 / 20

関数の多相性
リスト関数の多相性
リスト

いろいろな型のリスト

リスト関数の多相性

```
# [1,2,3];
val it = [1, 2, 3] : int list
# ["#a", "#b"]; ["abc"];
val it = ["#a", "#b"] : char list
val it = ["abc"] : string list
# [1, "a"]; (* 異なる型の要素はリストに出来ない *)
(interactive):35.0-35.7(362) Error:
  (type inference 023) operator and operand don't agree
  operator domain: ...
  operand: ...
# [[1,2], [3,4,5], [6]]; [(1, "a"), (2, "b")];
val it = [[1, 2], [3, 4, 5], [6]] : int list list
val it = [(1, "a"), (2, "b")] : (int * string) list
```

注意). `int list list` は, `(int list) list` の省略

リスト関数の多くは, どのような要素の型をもつリストであっても, 共通に用いることができる.

```
# rev [1,2,3,4];
val it = [4, 3, 2, 1] : int list
# rev ["aa", "bb", "cc"];
val it = ["cc", "bb", "aa"] : string list
#
```

関数`rev`の型を見てみると以下のようにになっている:

```
# rev;
val it = fn : ['a. 'a list -> 'a list]
```

つまり, 関数`rev`は, どんな型の要素をもつリストに対しても使える.

11 / 20

15 / 20

実習課題(1)

- 与えられたリストの最後の要素を返す関数 `last`. ただし、空リストに対してはエラーとなつてよい.


```
# last [1,2,3,4,5];
val it = 5 : int
```
- 自然数 n とリスト xs を受けとつて、 xs を n 回連結したリストを返す関数 `appendNtimes`. (ヒント: 再帰関数で定義する.)


```
# appendNtimes (3,[1,2]);
val it = [1, 2, 1, 2, 1, 2] : int list
```
- 正の整数 n を受けとつて、リスト $[n, n-1, \dots, 0]$ を返す関数 `natListDownFrom`.


```
# natListDownFrom 7;
val it = [7, 6, 5, 4, 3, 2, 1, 0] : int list
```

対のリストに関する組み込み関数

対のリストに関する組み込み関数が `ListPair` ストラクチャに入っている.

```
# ListPair.zip ([1,2,3],[4,5,6]);
val it = [(1, 4), (2, 5), (3, 6)] : (int * int) list
# ListPair.unzip [(1,2),(3,4),(5,6)]
val it = ([1, 3, 5], [2, 4, 6]) : int list * int list
```

- `ListPair.zip` は、リストの対 (xs, ys) を受けとつて、 xs と ys の要素を先頭から順に対応してできるリストを返す. 片方がもう片方より長い場合は、残りの要素は捨てられる.
- `ListPair.unzip` は、対のリストを受けとつて、対の第1要素からなるリストと、対の第2要素からなるリストを、対にして返す.

実習課題(1)

- 2つの整数 n, m ($n \leq m$ とする) をもらつて、リスト $[n, n+1, \dots, m]$ を返す関数 `intListFromTo`.


```
# intListFromTo (23,27);
val it = [23, 24, 25, 26, 27] : int list
```
- 文字列が回文になっているかを真理値で返す関数 `isPalindrom`.


```
# isPalindrom "rotator"; isPalindrom "hello";
val it = true : bool
val it = false : bool
```

実習課題(2)

- 整数 n と要素 a とリスト xs を受けとつて、リスト xs の n 番目の要素を a に置き替えたリストを返す関数 `replaceNth`.


```
# replaceNth (3,10,[1,2,3,4,5]);
val it = [1,2,3,10,5] : int list
```
- 整数 k とリスト $[a_1, a_2, \dots, a_n]$ を受けとつて、リスト $[a_{k+1}, a_{k+2}, \dots, a_n, a_1, \dots, a_k]$ を返す関数 `rotate`.


```
val rotate = fn : 'a. int * 'a list -> 'a list
# rotate (3,[1,2,3,4,5]);
val it = [4, 5, 1, 2, 3] : int list
```
- リスト $[a_1, a_2, \dots, a_n]$ を受けとつて、リスト $[(a_1, a_n), (a_2, a_{n-1}), \dots, (a_n, a_1)]$ を返す関数 `zipWithRev`.


```
# zipWithRev [1,2,3];
val it = [(1, 3), (2, 2), (3, 1)] : (int * int) list
```

リストに関する組み込み関数

リストに関する組み込み関数は、`List` ストラクチャにある.

```
# List.last [1,2,3];
val it = 3 : int
# List.nth ([1,2,3],1);
val it = 2 : int
# List.nth ([1,2,3],2);
val it = 3 : int
# List.take ([1,2,3],1);
val it = [1] : int list
# List.drop ([1,2,3],1);
val it = [2, 3] : int list
```

`last` に空リストを適用したり、`nth`、`take`、`drop` で指定がリストの範囲を越えていると、エラーとなる.

- `List.last` は、リストの最後の要素を返す.
- `List.nth` は、リスト xs と整数 n を受けとり、 xs の n 番目の要素を返す. なお、先頭の要素を0番目と数える.
- `List.take` は、リスト xs と整数 n を受けとり、 xs の先頭から n 個の要素からなるリストを返す.
- `List.drop` は、リスト xs と整数 n を受けとり、 xs の先頭から n 個の要素を取り除いたリストを返す.