

新しい型

胡振江

(東京大学計数工学科)



型演算子

型を構成する演算子:

- 関数型の構成

$T1 \rightarrow T2$

- 組型の構成

$(T1, T2, \dots, Tn)$

- リスト型の構成

$[T]$



本章の主題

あらたな型を直接に構築する仕組みを紹介する。

- 列挙型
- 複合型
- 再帰型
- 抽象型



列挙型

新しい型の値を列挙して数え上げる方法

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving (Eq, Ord)
```

```
workday :: Day -> Bool
workday d = (Mon <= d) && (d <= Fri)

weekday :: Day -> Bool
weekday d = (d==Sat) || (d==Sun)
```



複合型

- 値を並べて型を定義するだけではなくて、他の型に依存する値からなる型

データ構成子

```
data Tag = Tagn Int | Tagb Bool
```

値の例 : Tagn 10

Tagb True



- 関数定義の例

```
numVal :: Tag -> Int  
numVal (Tagn n) = n
```

```
isNum :: Tag -> Bool  
isNum (Tagn n) = True  
isNum (Tagb b) = False
```



多様型

- 型を型変数をパラメタとして定義する。

```
data Pair a b = Pair a b
```

例 : `Pair 3 True :: Pair Int Bool`

`Pair [3] 0 :: Pair [Int] Int`

`Pair (Pair 1 2) 3 :: Pair (Pair Int Int) Int`



型定義の一般形

```
data T a1 a2 ... an = ...  
    | Ci ti1 ti2 ... tiki  
    ...
```

型式 t

$t ::= v$

型変数

| $T t \dots t$

型構成式



再帰型

- 自然数

```
data Nat = Zero | Succ Nat
```

値の例 :

Zero

Succ (Zero)

Succ (Succ (Succ (Succ Zero)))



- 自然数上の関数定義

```
plus :: Nat -> Nat -> Nat
n `plus` Zero      = n
n `plus` (Succ m) = Succ (n `plus` m)
```

```
fibnat :: Nat -> Nat
fibnat Zero      = Zero
fibnat (Succ Zero) = Succ Zero
fibnat (Succ (Succ n)) = fibnat (Succ n) `plus` fibnat n
```

```
inf = Succ inf
```



再帰型

- リスト

```
data List a = Nil | Cons a (List a)
```

値の例 :

`Nil :: List a`

`Cons 1 (Cons 2 Nil) :: List Int`



再帰型

- 算術式

```
data Aexp = Num Int
          | Exp Aexp Aop Aexp
Data Aop = Add | Sub | Mul | Div
```

値の例 :

Exp (Num 3) Add (Num 4)

Exp (Num 3) Add (Exp (Num 4) Mul (Num 5))



- 算術式の評価

$\text{eval} :: \text{Aexp} \rightarrow \text{Int}$

$\text{eval} (\text{Num } n) = n$

$\text{eval} (\text{Exp } e1 \text{ op } e2) = \text{apply op} (\text{eval } e1) (\text{eval } e2)$

$\text{apply Add} = (+)$

$\text{apply Sub} = (-)$

$\text{apply Mul} = (*)$

$\text{apply Div} = (/)$



構造帰納法

```
data Aexp = Num Int
          | Exp Aexp Aop Aexp
Data Aop = Add | Sub | Mul | Div
```

- 場合 (Num n).
P (Num n)がすべてのnに対して成立すること。
- 場合 (Exp e1 op e2).
P(e1)とP(e2)が成立するならば、
P(Exp e1 op e2)がすべてのopに対して成立すること。



例：簡単なコンパイラの正しさの証明(1)

- 算術式を評価する簡単なスタック計算機を考える。

- 命令

- data Inst = Load Int | Apply aop

- 計算領域：スタック

- type Stack = [Int]

- 命令の実行

- execute :: Inst -> Stack -> Stack

- execute (Load x) xs = x : xs

- execute (Apply op) (x1:x2:xs) = apply op x2 x1 : xs

- 命令列の実行

- run :: [Inst] -> Stack -> Stack

- run [] xs = xs

- run (i:is) xs = run is (execute i xs)

run (in1++in2) xs

= run in2 (run in1 xs)



例：簡単なコンパイラの正しさの証明(2)

- コンパイラ：算術式を命令列に翻訳する。

`compile :: Aexp -> [Inst]`

`compile (Num n) = [Load n]`

`compile (Exp e1 op e2) =`

`compile e1 ++`

`compile e2 ++`

`[Apply op]`



例：簡単なコンパイラの正しさの証明(3)

- コンパイラの正しさ

$\text{run (compile } e) \text{ } xs = \text{eval } e : xs$

証明： A_{exp} 上の構造帰納法によって証明できる。



例：簡単なコンパイラの正しさの証明(3)

- コンパイラの正しさ

$\text{run (compile e) xs} = \text{eval e : xs}$

証明： 場合 (Num n).

$\begin{aligned} \text{run (compile (Num n)) xs} &= \text{run [Load n] xs} \\ &= \text{run [] (execute (Load n) xs)} \\ &= \text{execute (Load n) xs} \\ &= n : xs \\ &= \text{eval (Num n) : xs} \end{aligned}$



例：簡単なコンパイラの正しさの証明(3)

- コンパイラの正しさ

$\text{run (compile e) xs} = \text{eval e : xs}$

証明： 場合 (Exp e1 op e2).

$\begin{aligned} & \text{run (compile (Exp e1 op e2)) xs} \\ &= \text{run (compile e1 ++ compile e2 ++ [Apply op]) xs} \\ &= \text{run [Apply op] (run (compile e2) (run (compile e1) xs))} \\ &= \text{run [Apply op] (run (compile e2) (eval e1 : xs))} \\ &= \text{run [Apply op] (eval e2 : eval e1 : xs)} \\ &= \text{run [] (execute (Apply op) (eval e2 : eval e1 : xs))} \\ &= \text{execute (Apply op) [eval e2 : eval e1 : xs]} \\ &= \text{apply op (eval e1) (eval e2) : xs} \\ &= \text{eval (Exp e1 op e2) xs} \end{aligned}$



具象型と抽象型

具象型

例 : data List a = []
 | a : List a

値 : 構成子によって指定する

演算、関数 : 定められない

抽象型

例 : スタック
 push, pop, empty, top

演算 : 意味を指定する。
 満たすべき仕様を与える

値 : 定められない



抽象型の実装 : 値を決め、演算を定義する。

キュー (待ち行列) (1)

- Queue a
 - start :: Queue a
空のキューを生成する
 - join :: Queue a -> a -> Queue a
キューの後ろに要素を付加する
 - front :: Queue a -> a
キューの先頭要素を返す
 - reduce :: Queue a -> Queue a
キューの先頭要素を取り除く



キュー (待ち行列) (2)

- 代数的仕様記述

- $\text{front}(\text{join start } x) = x$
- $\text{front}(\text{join}(\text{join } q \ x) \ y) = \text{front}(\text{join } q \ x)$
- $\text{reduce}(\text{join start } x) = \text{start}$
- $\text{reduce}(\text{join}(\text{join } q \ x) \ y)$
 $= \text{join}(\text{reduce}(\text{join } q \ x)) \ y$



キュー (待ち行列) (3)

- リストによる実装

```
type Queue a = [a]
```

```
start = []
```

```
join q x = q ++ [x]
```

```
front = hd
```

```
reduce = tail
```

- 均衡木による実装

```
type Queue a = Btree a
```

```
start = EmptyTree
```

```
join q x = ...
```

ユーザから隠蔽する

Remark

- 期末試験
 - 2月7日 8:30-10:00
 - 教科書、ノート持ち込み可
- 復習について
 - 関数プログラム(の理解)
 - 関数プログラムの実行モデルと効率
 - プログラムの合成・導出
 - プログラムの性質の証明
 - 有限データ上のプログラム
 - 無限データ上のプログラム



質疑時間: 2月1日 14:00-18:00 6号館350室

Thank You!

