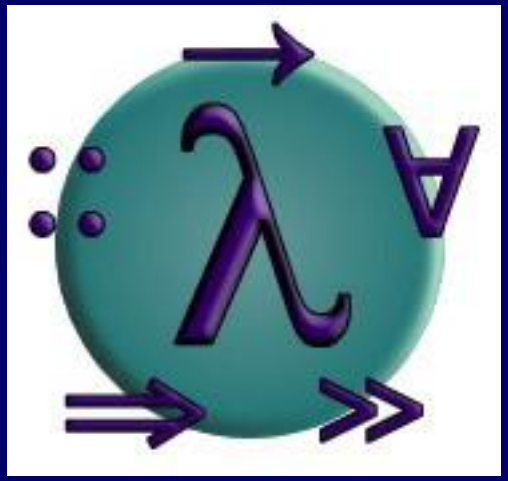
PROGRAMMING IN HASKELL
プログラミングHaskell



Chapter 13 – Reasoning about Programming
プログラムの論証

愛知県立大学 情報科学部 計算機言語論(山本晋一郎・大久保弘崇、2011年)講義資料
オリジナルは http://www.cs.nott.ac.uk/~gmh/book.html を参照のこと

等式推論

- 中学高校で学ぶ式の変形は、数と演算子に対する代数的性質を利用している
  - 例: 加算や乗算について成り立つ性質
    - x y = y x　　{* の交換則}
    - x + (y + z) = (x + y) + z　　{+ の結合則}
    - x (y + z) = x y + x z　　{* の + に対する左分配則}
    - (x + y) z = x z + y z　　{* の + に対する右分配則}
  - 性質を用いた論証の例
    - (x + a) (x + b)　　{左分配則}
      = (x + a) x + (x + a) b　　{右分配則}
      = x x + a x + x b + a b　　{* の交換則}
      = x^2 + a x + b x + a b　　{右分配則を右辺から左辺へ}
      = x^2 + (a + b) x + a b
  - (x + a) (x + b) と x^2 + (a + b) x + a b は等しいが、計算は前者の方が効率的(加算 2 回と乗算 1回)

Haskell に対する論証

- 同様に、Haskell のプログラムに対して成立する性質を論証しよう

- 利用者が与えた等式は性質として使える
  - 以下の定義
    ```
    double  :: Int → Int
    double x = x + x
    ```
    は、プログラムに関する論証で使用できる性質
  - 任意の Int 型の式 a に対して、double a を a + a に置き換えて良い
  - 左辺から右辺、右辺から左辺のどちらにも使える

Haskell に対する論証

- 複数の等式による関数定義に注意
  - 定義式の順序は重要
  - ```
    isZero :: Int → Bool
    isZero 0 = True        {常に使える}
    isZero n = False       {n ≠ 0 の場合のみ}
    ```

- ガード付き等式による等価な定義
  - 順序を気にしなくて良いので使いやすい
  - ```
    isZero 0               = True      {常に使える}
    isZero n | n ≠ 0       = False     {常に使える}
    ```

reverse と ++ の定義(復習)

```
reverse :: [a] → [a]                -- p.167
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]

(++) :: [a] → [a] → [a]             -- p.60
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Haskell に対する論証の例

- 要素が 1 つのリストを反転しても意味がない こと reverse [x] = [x] の証明
  -
```
    reverse [x]                 {リスト表記}
  = reverse (x:[])              {reverse の定義 2}
  = reverse [] ++ [x]           {reverse の定義 1}
  = [] ++ [x]                   {++ の定義 1}
  = [x]                         {よって成立、証明終了}
```

- プログラム中の reverse [x] は [x] と置き換えても良く、より効率的になる

整数に対する数学的帰納法

- 全ての有限な自然数に対して成り立つ性質を証明したい
  - 例: 全ての自然数 x に対して x + 0 = 0
  - 全部を数え上げることはできない
  - 数学的帰納法の出番

- ただし、数学の話ではなく、Haskell 上に定義した自然数を対象とする
  - 自然数の定義の例(10 章)

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

再帰型による自然数(10 章)の復習

再帰型

データ型は自分自身を使って定義することもできる。
すなわち、型は再帰的に定義されることもある。

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

Nat は 2 つの構成子 Zero :: Nat と
Succ :: Nat → Nat を持つ新しい型

Note:

- Nat 型の値は Zero か、あるいは Succ n の形をしている(ただし、n :: Nat)。すなわち、Nat は以下のような値の無限列を含む:

```
Zero

Succ Zero

Succ (Succ Zero)

⋮
```

- Nat 型の値は自然数とみなせる。すなわち、Zero は 0 を、Succ は 1 つ大きな整数を返す関数 (1+) を表している。

- 例えば、以下の値は、

```
Succ (Succ (Succ Zero))
```

自然数の 3 を表している

```
1 + (1 + (1 + 0))  =  3
```

再帰型による自然数(10 章)の復習終わり

add x Zero = x の証明

- 加算の定義

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

- 全ての自然数 x に対して、add x Zero = x が成り立つことを数学的帰納法で示してみる
  - x = Zero のとき

```
    add Zero Zero        {add の定義 1}
  = Zero                 {よって成立}
```

  - x = n のときに add n Zero = n が成り立つと仮定する

```
    add (Succ n) Zero    {add の定義 2}
  = Succ (add n Zero)    {帰納法の仮定}
  = Succ n               {よって成立、証明終了}
```

add x (add y z) = add (add x y) z の証明

- x = Zero のとき
  - 左辺 = add Zero (add y z)　{外側の add}
    = add y z
  - 右辺 = add (add Zero y) z　{内側の add}
    = add y z　{よって成立}
- x = n のときに add n (add y z) = add (add n y) z が成り立つと仮定する
  - 左辺 = add (Succ n) (add y z)　{外側の add}
    = Succ (add n (add y z))　{帰納法の仮定}
    = Succ (add (add n y) z)
  - 右辺 = add (add (Succ n) y) z　{内側の add}
    = add (Succ (add n y)) z　{外側の add}
    = Succ (add (add n y) z)　{よって成立、証明終了}

リストに対する数学的帰納法

- 整数に対する数学的帰納法
  - 全ての整数 x に対して P(x) が成立することを示す
    - 例: 全ての自然数 x に対して add x Zero = x
- リストに対する数学的帰納法
  - 全てのリスト xsに対して P(xs) が成立することを示す
    - 例: 全てのリスト xs に対して reverse (reverse xs) = xs
- 対応関係

	基底	帰納段階
整数	P(0) が成立	P(x) を仮定し、P(x+1) を示す
リスト	P([]) が成立	P(xs) を仮定し、P(x:xs) を示す

reverse (reverse xs) = xs の証明

- xs = [] のとき
  - rev (rev [])　{内側の rev の定義 1}
    = rev []　{rev の定義 1}
    = []　{よって成立}
- xs = ns のときに rev (rev ns) = ns が成り立つと仮定する
  - rev (rev (n:ns))　{内側の rev の定義 2}
    = rev (rev ns ++ [n])　{rev の ++ に対する分配則}
    = rev [n] ++ rev (rev ns)　{長さ 1 のリストの反転}
    = [n] ++ rev (rev ns)　{帰納法の仮定}
    = [n] ++ ns　{長さ 1 のリストの ++}
    = n:ns　{よって成立、証明終了}

長さ 1 のリストの反転	`rev [x] = [x]`
rev の ++ に対する分配則	`rev (xs ++ ys) = rev ys ++ rev xs`
長さ 1 のリストの結合	`[x] ++ xs = x:xs`

まとめ(12章)

- Haskell のプログラムの性質を論証する
  - (++) や reverse などのライブラリの性質は利用可能
    - 長さ 1 のリストの反転　　rev [x] = [x]
      rev の ++ に対する分配則　　rev (xs ++ ys) = rev ys ++ rev xs
    - 長さ 1 のリストの結合　　[x] ++ xs = x:xs
  - 利用者が与えた等式も性質として利用可能
    - 定義 `double x = x + x` が与えられたら
      - 任意の式 a に対して、double a を a + a に置き換えて良い
      - 左辺から右辺、右辺から左辺のどちらにも使える
- リストに対する数学的帰納法
  - 全てのリスト xs に対して P(xs) が成立することを示す
    - 例: 全てのリスト xs に対して reverse (reverse xs) = xs
  - データ構成子による場合分けが有用
    - 整数なら Zero と Succ x、リストなら [] と x:xs で場合分け

全体のまとめ(型の観点から、No.1)

- 基本型
  - Bool, Char, Int, Integer, Float
- リスト型(同じ型の値の並び (0個以上))
  - null, elem {述語}
  - length {長さ}
  - head, last, (!!) {要素の抽出}
  - tail, init, take, drop {部分リストの抽出}
  - (++), concat {連結}
  - zip {タプルのリスト}
  - map, filter {単純なリスト内包表記}
  - リスト内包表記
- タプル型(n 項組)
  - () {unit 型 (意味のない値を表す)}
  - fst, snd {要素の抽出}

全体のまとめ(型の観点から、No.2)

- 関数型
  - 関数定義方法
    - 条件文
    - 引数のパターンマッチ
    - ガード付き等式
    - 再帰定義
    - 畳込関数 (foldr, foldl)
    - リスト内包表記 (map, filter)
  - ラムダ記法 (値としての関数を表す記法)
  - カリー化 (Haskell の関数は 1 引数)
  - 多相型 (型変数を含む型)
  - 多重定義型 (型のテンプレート)

全体のまとめ(型の観点から、No.3)

- 利用者定義の型
  - type 宣言 (別名)
  - data 宣言 (新しい型を定義)
    - データ構成子
- クラス (型のテンプレート)
  - Eq (同等)
  - Ord (順序)
  - Show (文字列化), Read (読み込み可能)
  - Num (数値，除算なし)
  - Integral (整数，商と余り), Fractional (分数，除算と逆数)
  - Monad (return と bind を備える)

全体のまとめ(型の観点から、No.4)

- その他の重要な型
  - Maybe a
    - エラーを考慮した値を表す型
    - `type Maybe a = Just a | Nothing`
  - Parser a
    - a 型の値を返すパーサー
    - `type Parser a = String → [(a,String)]`
      - 読み込んだ a と読み残した String のタプルを返す
      - 失敗:　　空リスト
      - 成功(受理): 要素が 1 つのリスト(singleton list)
  - IO a
    - 副作用を持つ型
    - 入出力を表す