Haskellを利用した項書換え系の実現

草刈圭一朗，酒井正彦

初版：2005年3月31日
改訂：2010年5月10日（西田直樹）
改訂（Haskell版）：2021年4月12日（橋本健二）

本資料では，Haskellを用いて項書換え系を実現するための方針について解説する．なお，項書換え系に関する諸概念を理解するために必要となる記法や基本概念を付録に紹介しておいたので参考にされたし．

1　汎用ユーティリティ（Utilモジュール）

本節ではプログラム中で用いる基本関数を定義しておく`Util`モジュールに関しての解説をする．最低限定義しておいて欲しい関数として以下の関数を`Util`モジュールに加えておいて貰いたい．もちろん，他の構造のプログラミング時の必要性に応じて必要な関数はどんどん追加していくべし．

```
module Util where

type Assoc k a = [(k,a)]
type Set a = [a]

find   :: Eq k => k -> Assoc k a -> Maybe a
append :: (Eq k, Eq a) => Assoc k a -> Assoc k a -> Maybe (Assoc k a)
union        :: Eq a => Set a -> Set a -> Set a
intersection :: Eq a => Set a -> Set a -> Set a
```

なお，上記の内容は配布した`Util.hs`の中身である．`Util.hs`に書き加えることにより`Util`モジュールを完成させて欲しい．以下では，各関数の仕様を記す．

連想リストの型`Assoc k a`は`k`の型のデータと`a`の型のデータの対のリストであり，`k`の型のデータと対応する`a`の型のデータの情報を保持する．連想リストの型のデータ`xs`は次の性質を満たすと仮定しておく．

$$\forall (l1, v1), (l2, v2) \in \texttt{xs}.\ [l1 = l2 \Rightarrow v1 = v2]$$

連想リスト型のデータを取り扱う関数として，与えられた`k`の型のデータと連想リストから対応する`a`の型のデータの情報を引き出す関数`find`と，連想リストを連接する関数`append`を用意する．

$$\texttt{find}\ l\ xs = \begin{cases} \texttt{Just}\ v' & \text{if } \exists (l, v') \in xs \\ \texttt{Nothing} & \text{otherwise} \end{cases}$$

$$\texttt{append}\ xs\ ys = \begin{cases} \texttt{Just}\ zs & \text{if } zs = xs \cup ys \text{ が連想リスト} \\ \texttt{Nothing} & \text{otherwise} \end{cases}$$

関数`union`と`intersection`は以下の関係を満たすように定義されたし．ただし，入力のリストxsとysが共に$[\ldots, x, \ldots, x, \ldots]$のように重複して出現する要素を含まない場合，返し値も要素が重複しないように定義すること．

$$\begin{aligned} \texttt{union}\ xs\ ys &= xs \cup ys \\ \texttt{intersection}\ xs\ ys &= xs \cap ys \end{aligned}$$

2　基本データの表現 (TRS モジュール)

変数, 項, 書き換え規則, 等式などは配布した TRS.hs の中の TRS と名付けられた構造 (structre) で以下のような型を用いて表現する旨が宣言されている．

```
data Symbol = VSym (String, Int) | FSym String
data Term = Node (Symbol, [Term])
type Pattern    = Term
type Rule       = (Pattern, Pattern)
type RuleSet    = [Rule]
type Equation   = (Pattern, Pattern)
type EquationSet= [Equation]
type Subst = Assoc (String, Int) Term
```

Symbol 型は関数記号と変数記号を，Term 型は項を，Pattern 型は書き換え規則や等式に出現する項を，Rule 型は規則を，RuleSet 型は規則集合を，Equation 型は等式を，EquationSet 型は等式集合を，Subst 型は代入を表現する．簡便のため，小文字で始まる名前を変数とし，それ以外の名前は関数記号とする．書き換え規則や等式は項の対で表す．書き換え規則の集合や等式の集合はそれらのリストで表す．ここで混乱を避けるために Term 型の別名として Pattern 型を定義してある．Pattern 型は書き換え規則や等式に出現する項に対して用いる．今回の定義では，項 $F(A, x)$ は Term 型で

```
Node (FSym "F", [Node (FSym "A", []), Node (VSym ("x",n), [])])
```

と内部的には表現される．ここで，注意して欲しいのは変数 x が，VSym ("x",n) と表されていることである．ここで，n はなんらかの自然数であり，変数の名前変えに対応するためのデータである (詳細は構造 Rename の説明のところで述べる)．実際に上記の項（$n = 0$ とする）を変数 t に入れてみよう．

```
> t = Node (FSym "F", [Node (FSym "A", []), Node (VSym ("x",0), [])])
> t
F(A, x)
```

TRS モジュールで定義した項などのデータを，そのままの形で入出力を行うのは大変である．よって，入出力ルーチンが用意してある．データ読み込み用の関数は TRS モジュールにおいて定義されている．各関数の型は以下のように宣言されている．

```
rdterm :: String -> Term
rdrule :: String -> Rule
rdrules:: [String] -> RuleSet
rdeq   :: String -> Equation
rdeqs  :: [String] -> EquationSet
```

データをプリティープリント用に文字列に変換する関数も配布した TRS.hs のファイルの中で定義されている．各関数の型は以下のように宣言されている．

```
prsym  :: Symbol -> String
prterm :: Term -> String
prrule :: Rule -> String
prrules:: RuleSet -> String
preq   :: Equation -> String
preqs  :: EquationSet -> String
```

利用法は以下の通りである．

```
> t = rdterm "F(x)"
> prterm t
"F(x)"
> r = rdrule "F(x) -> G(x)"
> prrule r
"F(x) -> G(x)"
> rs = rdrules ["F(x) -> G(x)", "A -> B"]
> prrules rs
"[ F(x) -> G(x),\n  A -> B ]\n"
> e = rdeq "F(x) = G(x)"
> preq e
"F(x) = G(x)"
> es = rdeqs ["F(x) = G(x)", "A = B"]
> preqs es
"[ F(x) = G(x),\n  A = B ]\n"
> sb = [(("x",0), rdterm "S(0)"), (("y",1), rdterm "0"), (("z",0), rdterm "P(x)")]
> prsubst sb
"[ x := S(0),\n  y:1 := 0,\n  z := P(x) ]\n"
```

最後に，項のサイズ (項に出現する記号の総数) を計算する関数 tsize の定義を例題として記しておく．

```
tsize (Node (f, ts)) = 1 + tsizelist ts
tsizelist [] = 0
tsizelist (t:ts) = tsize t + tsizelist ts
```

なお，項を表現するデータ型 Term は木構造を持ち，それゆえに 2 次元的 (縦方向と横方向) な再帰構造を持つ．よって，上述の定義のように縦方向の再帰用の関数 (tsize) と，横方向の再帰用の関数 (tsizelist) を別々に与え相互再帰で定義するとプログラミングしやすい場合が多々ある．

問 1 以下の関数を定義し，TRS モジュールに追加せよ．

1. 項 t を入力すると，項の深さ ($Pos(t)$ における最大長の文字列の長さ) を計算する関数 depth.
2. 項 t を入力すると，その項に出現する変数全体のリスト ($Var(t)$ に対応) を返す関数 varlist.

3 項書換え系の実現（Reduce モジュール）

書き換え規則は Rule 型，すなわち Pattern * Pattern 型として，項書換え系は RuleSet 型，すなわち Rule 型のリストとして実現する．これらの型は TRS モジュールで定義されている．Reduce モジュールでは，以下の関数を定義することにより項書換え系を実現する．

```
module Reduce where

  import Util
  import TRS

  subst :: Subst -> Term ->  Term

  match :: Pattern -> Term -> (Bool, Subst)
  matchlist :: [Pattern] -> [Term] -> (Bool, Subst)

  rewrite :: RuleSet -> Term -> (Bool, Term)

  {- Par-Out one-step reduction -}
  postep :: RuleSet -> Term -> (Bool, Term)

  {- Get normal form by Par-Out strategy -}
```

```
ponf :: RuleSet -> Term -> Term

{- Left-Out one-step reduction -}
lostep :: RuleSet -> Term -> (Bool, Term)

{- Get normal form by Left-Out strategy -}
lonf :: RuleSet -> Term -> Term

{- Get normal form by Left-In strategy -}
linf :: RuleSet -> Term -> Term
```

3.1 代入

代入は，TRS モジュールで定義した Subst 型，すなわち変数名に対応する `string * int` 型と Term 型のデータの連想リストで実現する．`subst` $a\ t$ は項 t の中の変数を対応する項に置き換える．変数と項の対応関係は連想リスト a に従う．

```
> s = [(("x",0), rdterm "0"), (("y",0), rdterm "S(0)")]
> t = rdterm "F(G(x),y)"
> t' = subst s t
> prterm t'
"F(G(0),S(0))"
```

実装上は `x := x` の場合も許すこととする．

3.2 照合

規則 $l \to r$ の左辺 l が項 t に照合するとは，ある代入 θ が存在して $t \equiv l\theta$ となることである．関数 `match` は，`Pattern` 型の式 l と `Term` 型の式 t を入力し照合の正否と代入を対にした値を返す以下の様な関係を満たす関数である[1]．

$$\texttt{match}\ l\ t \quad = \quad \begin{cases} (\mathit{True},\ \theta) & \text{if } \exists\theta.\ l\theta \equiv t \\ (\mathit{False}, [\,]) & \text{otherwise} \end{cases}$$

関数 `match` を定義するとき，以下の用な関係を満たす関数 `matchlist` を相互再帰で同時に定義しておくとプログラミングしやすいであろう．

$$\texttt{matchlist}\ [l_1, \ldots, l_n]\ [t_1, \ldots, t_m] \quad = \quad \begin{cases} (\mathit{True},\ \theta) & \text{if } n = m \text{ かつ } (\exists\theta.\ \forall i.\ l_i\theta \equiv t_i) \\ (\mathit{False}, [\,]) & \text{otherwise} \end{cases}$$

`matchlist` は以下のような型で定義してもよい（その際には `matchlist` の型宣言も変更すること）．

```
matchlist :: Subst -> [Pattern] -> [Term] -> (Bool, Subst)
```

$$\texttt{matchlist}\ \sigma\ [l_1, \ldots, l_n]\ [t_1, \ldots, t_m] \quad = \quad \begin{cases} (\mathit{True},\ \sigma \cup \theta) & \text{if } n = m \text{ かつ } (\exists\theta.\ \forall i.\ l_i\theta \equiv t_i) \\ & \quad \text{かつ} \sigma \cup \theta \text{が連想リスト} \\ (\mathit{False}, [\,]) & \text{otherwise} \end{cases}$$

3.3 先頭がリデックスならば 1 ステップ書き換える関数

`rewrite` $rs\ t$ は，項 t と照合する最初の rs の規則を用いて 1 ステップだけ書き換える．返し値はブール値と項の対であり，関数 `match` や `matchlist` のときと同様に対の第一要素は書き換えの正否を表現する．

$$\texttt{rewrite}\ rs\ t \quad = \quad \begin{cases} (\mathit{True},\ t') & \text{if } \exists(l, r) \in rs.\ \exists\theta.\ t \equiv l\theta \text{かつ } t' \equiv r\theta \\ (\mathit{False},\ t) & \text{otherwise} \end{cases}$$

[1] Maybe 型を用いて定義しても良い．今回は様々なプログラミングを習得して貰うために，あえてこの定義を採用している．

3.4 正規形を得るための関数

さて，準備は全て整った．項書換え系と項を与えると，その正規形を並列最外戦略・最左最外戦略・最左最内戦略でそれぞれ求める関数 `linf`, `ponf`, `lonf` を定義しよう．

- `linf` rs t
  最左最内戦略によって項 t の正規形を求める．以下の順で定義すれば良いだろう．

  1. `linf` を使ってすべての直接の部分項[2]を正規形に置き換えて t から t' を求める．
  2. ∗ t' 自身がリデックスのとき，t' を書き換え，その結果に `linf` を適用する[3]．
     ∗ そうでないとき，t' を返す．

  Haskell の遅延評価により，以上の手順を実装したつもりでもそのように実行されない場合がある．演算子 `$!` や `seq` を用いた正格適用を利用して，部分項を正規形に完全に置き換える前に次の処理に進まないようにすること．

- `postep` rs t
  項 t を並列最外戦略で 1 ステップだけ書き換える[4]．リデックスが 1 つも存在しないときのみ偽となるブール値と書き換え結果の項の対を返す．以下のように定義すれば良いだろう．

  – t 自身がリデックスのとき，t を書き換えてリターンする．
  – そうでないとき，すべての直接の部分項に対して `postep` を適用した結果を返す．

  なお，関数 `match` の定義のとき関数 `matchlist` を同時に相互再帰で定義したように，関数 `postep` に対する関数 `posteplist` を相互再帰で定義するとプログラミングしやすい．

- `ponf` rs t
  並列最外戦略により項 t の正規形を求める．関数 `postep` を用いる．

- `lostep` rs t
  項 t の最左最外リデックスを 1 ステップだけ書き換える．リデックスが存在しないときのみ偽となるブール値と書き換え結果の項の対を返す．以下のように定義すれば良いだろう．

  – t 自身がリデックスのとき，t を書き換えてリターンする．
  – そうでないとき，書き換えに成功するまで `lostep` を直接の部分項にたいし左から順に適用する．

  なお，関数 `postep` の場合と同様に，関数 `lostep` に対する関数 `losteplist` を相互再帰で定義するとプログラミングしやすい．

- `lonf` rs t
  最左最外戦略によって項 t の正規形を求める．関数 `lostep` を用いる．

4 問題

問 2 以下の性質を持つ項書換え系と項を見つけ，各自の作成したシステムで検証せよ．

(i) 最左最内戦略では正規形が求まるが，最左最外戦略や並列最外戦略では求まらない．

(ii) 最左最外戦略や並列最外戦略では正規形が求まるが，最左最内戦略では求まらない．

(iii) 並列最外戦略では正規形が求まるが，最左最外戦略では求まらない．

[2] 項 $F(t_1, \ldots, t_n)$ において $t_1, \ldots, t_n$ を直接の部分項 (immediate subterm) と呼ぶ．
[3] この部分を変更することにより効率化が可能である．t' に書き換える際の代入を変更し，代入の際，変数以外の部分だけに 2. が適用されるようにする．付録 B 参照．
[4] 並列最外戦略での 1 ステップの書き換えは，一般に数ステップの書き換えに対応することに注意．

参考文献

[1] F.Baader, T.Nipkow, Term Rewriting and All That, Cambridge University Press (1998).

[2] R.Bird, P.Wadler 著, 武市正人訳, 関数プログラミング, 近代科学社, 1991. (ISBN 4-7649-0181-1)

[3] Graham Hutton 著, 山本 和彦 訳, プログラミング Haskell 第 2 版, 技術書出版ラムダノート, 2019. (ISBN: 978-4908686078)

A　項書換え系

本節では項書換え系に関する諸定義を簡単に与える．詳細は文献 [1] を参照されたし．

定義 A.1 (項) 関数記号の集合を Σ で記し，変数記号の集合を V で記す．ここで $V \cap \Sigma = \emptyset$ であるとする．各関数記号 F は固有の引数個数を持ち，これを $arity(F)$ で記す．項 *(Term)* を以下のように帰納的に定義し，項の全体からなる集合を $T(\Sigma, V)$ で記す．

- 変数 $x \in V$ は項である．
- $F \in \Sigma$ かつ $arity(F) = n$ かつ $t_1, t_2, \ldots, t_n$ が項ならば $F(t_1, t_2, \ldots, t_n)$ も項である．

特に $arity(A) = 0$ となる関数記号を定数記号と呼び，通例に従い項 $A()$ は単に A と記される．また，2 つの項 s と t が等しいことを $s \equiv t$ で表し，項 t に出現する変数の全体からなる集合を $Var(t)$ で表す．

定義 A.2 (位置) 項 t における位置の集合 $Pos(t)$ を正整数の列 *(*空列を ε で表現*)* を用いて以下のように定義する．

$$Pos(t) = \begin{cases} \{\varepsilon\} & \text{if } t \in V \\ \{\varepsilon\} \cup \{iu \mid 1 \leq i \leq n,\ u \in Pos(t_i)\} & \text{if } t \equiv F(t_1, \ldots, t_n) \end{cases}$$

u, v をある項における位置とする．ある正整数列 w が存在して $u = vw$ となるとき，u は v より内側の位置であるといい，v は u より外側の位置であるという．ある整数 i, j $(i < j)$ と正整数列 w, u', v' が存在して $u = wiu'$ かつ $v = wjv'$ となるとき，u は v より左側の位置であるといい，v は u より右側の位置であるという．

定義 A.3 (文脈) 文脈 *(context)* とはホールと呼ばれる特別な定数記号 $\Box$ を一つだけ含む[5]項である．ホール自身も文脈であり，このような文脈を空の文脈と呼ぶ．文脈 $C[\,]$ に出現する $\Box$ を項 t で置き換えることによって得られる項を $C[t]$ で記す．特に，文脈 $C[\,]$ におけるホール $\Box$ の出現位置 p を明記したいときは $C[\,]_p$ のように添字 p で明示する．

$C[\,]$ を文脈 $F(G(\Box), H(y))$ とし t を項 $H(x)$ とすると，$C[t] \equiv F(G(H(x)), H(y))$ となる．

定義 A.4 (代入) 代入 *(substitution)* とは変数から項への写像である．代入 θ に対して項上の代入 $\hat{\theta}$ を以下で再帰的に定義する．

$$\hat{\theta}(t) = \begin{cases} \theta(x) & \text{if } t \equiv x \in V \\ F(\hat{\theta}(t_1), \ldots, \hat{\theta}(t_n)) & \text{if } t \equiv F(t_1, \ldots, t_n) \text{ かつ } F \in \Sigma \end{cases}$$

以下では通例に従い代入 θ と対応する項上の代入 $\hat{\theta}$ を同一視する．また，$\theta(t)$ を $t\theta$ で記す．

[5] 一般には複数のホールを含む文脈を考える．

定義 A.5 (照合, 例) 項 s, t に対し，ある代入 θ が存在して $s\theta \equiv t$ となるとき，s は t に照合 *(match)* するといい，t は s の例 *(instance)* であるという．

定義 A.6 (書き換え規則) 書き換え規則 *(rule)* とは項の対 (l, r) で変数条件，$l \notin V$ かつ $Var(l) \supseteq Var(r)$，を満たすものである．以下では $l \to r$ で記す．

定義 A.7 (項書換え系) 項書換え系 *(term rewriting system)* とは書き換え規則の集合である．項書換え系 R における書き換え関係 $\underset{R}{\to}$ を以下で定義する．

$$s \underset{R}{\to} t \iff^{\text{def}} \exists l \to r \in R,\ \exists C[\,]_p,\ \exists \theta,\quad s \equiv C[l\theta]_p \text{かつ } t \equiv C[r\theta]_p$$

特に $p = \varepsilon$ のとき，根の位置での書き換えという．また，上記の定義において項 $s \equiv C[l\theta]_p$ の部分項 $l\theta$ を位置 p におけるリデックス (redex) と呼ぶ．また，リデックスを含まない項を正規形 (normal form) と呼ぶ．文脈から R が明らかなときには $\underset{R}{\to}$ を $\to$ で略記する．

定義 A.8 (戦略) 最内リデックスのうち最も左に出現するものを書き換えていく戦略を最左最内戦略と呼ぶ．最外リデックスのうち最も左に出現するものを書き換えていく戦略を最左最外戦略と呼ぶ．全ての最外リデックスを一斉に書き換えていく戦略を並列最外戦略と呼ぶ．

例 A.9 次の項書換え系 R は項 $0, S(0), S(S(0)), \cdots$ を自然数 $0, 1, 2, \cdots$ とみなした場合，項書換え系による自然数上の加算と乗算の実現となっている．

$$R = \left\{ \begin{array}{rcl} Add(0, y) & \to & y \\ Add(S(x), y) & \to & S(Add(x, y)) \\ Mul(0, y) & \to & 0 \\ Mul(S(x), y) & \to & Add(Mul(x, y), y) \end{array} \right.$$

項 $Mul(S(0), Add(0, S(0)))$ を最左最内戦略 $\underset{li}{\to}$ と最左最外戦略 $\underset{lo}{\to}$ と並列最外戦略 $\underset{po}{\to}$ によりそれぞれ書き換えた書き換え列は以下のようになる．

$$\begin{array}{rcl} Mul(S(0), \underline{Add(0, S(0))}) & \underset{li}{\to} & \underline{Mul(S(0), S(0))} \\ & \underset{li}{\to} & Add(\underline{Mul(0, S(0))}, S(0)) \\ & \underset{li}{\to} & \underline{Add(0, S(0))} \\ & \underset{li}{\to} & S(0) \end{array}$$

$$\begin{array}{rcl} \underline{Mul(S(0), Add(0, S(0)))} & \underset{lo}{\to} & Add(\underline{Mul(0, Add(0, S(0)))}, Add(0, S(0))) \\ & \underset{lo}{\to} & \underline{Add(0, Add(0, S(0)))} \\ & \underset{lo}{\to} & \underline{Add(0, S(0))} \\ & \underset{lo}{\to} & S(0) \end{array}$$

$$\begin{array}{rcl} \underline{Mul(S(0), Add(0, S(0)))} & \underset{po}{\to} & Add(\underline{Mul(0, Add(0, S(0)))}, \underline{Add(0, S(0))}) \\ & \underset{po}{\to} & \underline{Add(0, S(0))} \\ & \underset{po}{\to} & S(0) \end{array}$$

B　最左最内戦略による評価の高速化

本文でも少し触れたが最左最内戦略によって正規形を求める関数 `linf` の高速化について解説する．先ず，`linf` の定義は以下のように変更する．

- `linf` rs t
  最左最内戦略によって項 t の正規形を求める．以下の順で定義すれば良いだろう．

  1. `linf` を使ってすべての直接の部分項を正規形に置き換えて t から t' を求める[6]．
  2. `linftop` rs t' の値を返す．

上記の `linf` の定義において用いた関数 `linftop` は関数 `lisubst` と共に相互再帰で定義する．なお，この二つの補助関数は `linf` の定義の内部 (let 文を用いる) で定義されたし．これらの関数は以下の性質を満たすように定義すれば良い．

- `linftop` rs t
  $t \equiv l\theta$ となる規則 $l \to r \in rs$ と代入 θ が存在した場合は `lisubst` θ r の値を返す．そうでない場合は t を返す．

- `lisubst` θ t
  t が変数の場合は $t\theta$ を返す．そうでない場合は以下のように定義する．

  1. `lisubst` θ を使って t の全ての直接の部分項を評価した値を t' とする．
  2. t' に `linftop` rs を適用．なお，この rs は `linf` に最初に入力されていた rs を用いる．

デバッグとして TRS $\{ F(S(x)) \to S(x) \}$ を用いて，項 $F(S(F(S(0))))$ を書換えてみること．正しい正規形は $S(S(0))$ であるが，`linf` を `linftop` に間違えていると結果は $S(F(S(0)))$ になってしまう．

さらなる高速化の方法として，被定義記号と構成子を区別する方法がある．

定義 B.1 (被定義記号，構成子) 項書換え系 R の左辺の根の位置に現れる関数記号を R **の被定義記号**と呼ぶ．さらに，R の被定義記号でない関数記号を R **の構成子**と呼ぶ．構成子と変数だけから構成される項は R **の構成子項**と呼ばれる．関数記号の集合を Σ，R の被定義記号の集合を Σ_D，R の構成子の集合を Σ_C とすると，Σ_D と Σ_C は次のように定義される．

$$\Sigma_D = \{ f \mid f(t_1, \ldots, t_n) \to r \in R \}$$
$$\Sigma_C = \Sigma \setminus \Sigma_D$$

リデックスの根記号は被定義記号であることから，項がリデックスかどうかを判定する際に，注目している項の根記号が構成子であれば照合を行わなくてもリデックスでないことがわかる．この方法は，被定義記号の集合をあらかじめ求めておき，それを各関数の引数で持ち回すことで容易に実現できる．

[6] `linf` を `linftop` と間違えないように注意すること．`linf` と `linftop` は型は一致するため，間違ったことに気付かない場合がある．このような場合のバグを発見するために，必ず動作テストを行わなければならない．