プログラマのための代数構造

Ichi Kanaya

2016

目次

第 I 部

代数構造とプログラミング

第 1 章

はじめに

Haskell というプログラミング言語を知ろうとすると，従来のプログラミング言語の知識が邪魔をする．モダンで，人気があって，Haskell から影響を受けた言語，例えば Ruby や Swift の知識さえ，Haskell を学ぶ障害になり得る．ではどのようにして Haskell の深みに到達すればいいのだろうか．

その答えは，一見遠回りに見えるが，一度抽象数学の高みに登ることである．

と言っても，あわてる必要はない．

近代的なプログラミング言語を知っていれば，すでにある程度抽象数学に足を踏み入れているからである．

第 2 章

数学的準備

a

2.1 数・体・環

これから各種の代数的構造を見ていくことにする．代数的構造と言っても，身構える必要はない．それは，我々プログラマが日々接している概念に，共通した名前を与えたにすぎない．

まず最初に，我々にとって一番身近な代数構造である**数**を見てみよう．数の代表例は**実数**であるから，実数を例にとって考えてみよう．実数全体の集合を $\boldsymbol{R}$ で表すことにする．また任意の実数を x, y, z で表すこととする．このことを数学者は $x, y, z \in \boldsymbol{R}$ と書くが，我々は記号 $\in$ を別の用途に使いたいので，本書では

$$x, y, z :: \boldsymbol{R}$$

と表すことにする．

以下に実数の備える代数的性質を列挙する．どれも当たり前のことに見えるが，ひとつひとつ見ていこう．ここで $x, y, z :: \boldsymbol{R}$ とする．

性質 1. x と y の足し算（**加法**）の結果（**和**）$x + y$ は $\boldsymbol{R}$ の元すなわち実数である．

性質 2. **足し算の結合則** x, y, z について

$$(x + y) + z = x + (y + z)$$

である．これを**結合則**と呼ぶ．

性質 3. **零元（加法単位元）の存在** $0 :: \boldsymbol{R}$ があり

$$0 + x = x + 0 = x$$

である．この 0 は足し算の**単位元**である．**零元**または**加法単位元**と呼ぶこともある．

性質 4. **負元（加法逆元）の存在** x に対して $-x :: \boldsymbol{R}$ があり

$$-x + x = 0$$

である．この $-x$ は x の足し算の**逆元**である．**負元**または**加法逆元**と呼ぶこともある．

性質 5. **足し算の可換性** x, y について

$$x + y = y + x$$

である．このことを足し算の**可換性**と呼ぶ．

性質 6. **掛け算** x と y の掛け算（**乗法**）の結果（**積**）$x * y$ は $\boldsymbol{R}$ の元すなわち実数である．

性質 7. **掛け算の結合則** x, y, z について

$$(x * y) * z = x * (y * z)$$

である．

性質 8. **単位元の存在** $1 :: \boldsymbol{R}$ があり

$$1 * x = x * 1 = x$$

である．この 1 を掛け算の単位元または**乗法単位元**と呼ぶ．

性質 9. **逆元** x に対して $x^{-1} :: \boldsymbol{R}$ があり

$$x^{-1} * x = 1$$

である．この x^{-1} は x の掛け算の逆元である．**乗法逆元**と呼ぶこともある．ただし性質 12 で述べる通り，加法単位元については逆元がなくても良い．

性質 10. **掛け算の可換性** x, y について

$$x * y = y * x$$

である．このことを掛け算の可換性と呼ぶ．

性質 11. **分配則** 足し算と掛け算が混在する場合

$$x * (y + z) = x * y + x * z$$

と掛け算を**分配**する．

性質 12. **加法単位元の逆元** 加法単位元に対する掛け算の逆元は存在しなくても良い．（つまり 0^{-1} のことは考えなくて良い．）

以上が実数の代数的性質の全てである．我々がよく使う割り算，引き算はプログラミングでいう糖衣構文 (syntax sugar) である．

上述の 12 個の条件が当てはまる数には，**有理数**や**複素数**がある．この 12 個の性質をまとめて，数学では**体**または**可換体**と呼ぶ．体の要素は，集合 $\boldsymbol{K}$，二項演算子 $+$，二項演算子 $+$ の単位元 0，二項演算子 $+$ の逆元生成演算子 $-$，もう一つの二項演算子 $*$，二項演算子 $*$ の単位元 1，二項演算子 $*$ の逆元生成演算子 $^{-1}$ であるから，体はそれらを列挙して $(\boldsymbol{K}, +, 0, -, *, 1, ^{-1})$ と表現する．

一方で，この数の性質を若干緩めたい場合もある．さもなければ，**整数**，**正則行列**（逆行列の存在する行列），**クォータニオン**，**論理値**，**ベクトル**，ベクトルの**変換**，**写像**と言った重要な概念が数の概念からこぼれてしまうからである．例えば整数の場合は掛け算の逆元は（単位元の逆元を除いて）存在しないし，正則行列やクォータニオンの場合は掛け算が可換ではない．前者は**可換環**（掛け算が非可換な場合は単に**環**），後者は**斜体**といった，それぞれ別な性質名が与えられている．

制約を少しずつ緩める代わりに，制約をその構成要素に分解することもできそうである．体には二つの二項演算子 $+$ と $*$ が登場した．その片方にのみ注目してみたらどうなるだろう．それが次節で取り上げる**群**である．

2.2　群

いま $x, y, z :: \boldsymbol{G}$ とし，二項演算子を形式的に $\bigstar$ と書くことにしてみよう．

性質 1.　x と y の演算の結果 $x \bigstar y$ は $\boldsymbol{G}$ の元である．

性質 2.　x, y, z について

$$(x \bigstar y) \bigstar z = x \bigstar (y \bigstar z)$$

である．これを結合則と呼ぶ．

性質 3.　$O :: \boldsymbol{G}$ があり

$$O \bigstar x = x \bigstar O = x$$

である．この O は単位元である．

性質 4.　x に対して $\tilde{x} :: \boldsymbol{G}$ があり

$$\tilde{x} \bigstar x = O$$

である．この $\tilde{x}$ は x の逆元である．

性質 5.　x, y について

$$x \bigstar y = y \bigstar x$$

である．

このような性質が満たされる時，組み合わせ $(\boldsymbol{G}, O, \bigstar)$ を**可換群**または**加群**と呼ぶ．例えば $(\boldsymbol{R}, 0, +)$ は可換群である．整数全体の集合を $\boldsymbol{Z}$ とすると $(\boldsymbol{Z}, 0, +)$ も可換群である．また，集合 $\boldsymbol{R}$ から 0 だけを取り除いた集合を $\boldsymbol{R} \setminus 0$ とするとき $(\boldsymbol{R} \setminus 0, 1, *)$ も可換群である．

可換群は代表的な**代数構造**のひとつであり，他にも数学のあちこちに顔を出している．例えば回転角を r とする二次元の回転変換（行列）を U_r ただし

$$U_r = \begin{bmatrix} \cos r & -\sin r \\ \sin r & \cos r \end{bmatrix}$$

として，回転変換 U_r すべてからなる集合 $\boldsymbol{U}$ を考えてみよう．回転の合成を $\bullet$ で表すとすると

$$U_{r_1} \bullet U_{r_2} = U_{(r_1 + r_2)} \tag{2.1}$$

であるから，回転を合成した結果も回転である．また式 (2.1) から

$$U_{r_1} \bullet (U_{r_2} \bullet U_{r_3}) = (U_{r_1} \bullet U_{r_2}) \bullet U_{r_3}$$

であるから，回転変換は結合則も満たしている．

次にに回転変換に単位元があるかどうか調べてみよう．何もしない回転変換は 0 度の回転であるから，何もしない回転を U_0 とよう．そうすると

$$U_0 \bullet U_r = U_r \bullet U_0 = U_r$$

であるから，U_0 は回転変換の単位元であると言える．

最後に回転変換に逆元があるかも調べてみよう．r 回転の逆は明らかに $-r$ であるから

$$U_{-r} \bullet U_r = U_r \bullet U_{-r} = U_0$$

が成り立つ．このように，組み合わせ $(\boldsymbol{U}, U_0, \bullet)$ も可換群である．

ところで，可換群の性質のうち最初の 4 項目だけを満たすものを**群**と呼ぶ．可換群は群の特別な場合である．現代の数学では $x \bigstar y \neq y \bigstar x$ のように演算子の前後を入れ替えると結果が異なるような演算をよく取り扱うので，一般の群は可換群よりもよく取り上げられ，それ故より短い名前が付けられている．

2.3 モノイド

集合 $\boldsymbol{G}$ が群である条件は

群の性質 1. x と y の演算の結果 $x\bigstar y$ は $\boldsymbol{G}$ の元である.

群の性質 2. x,y,z について

$$(x\bigstar y)\bigstar z = x\bigstar(y\bigstar z)$$

である. これを結合則と呼ぶ.

群の性質 3. $O :: \boldsymbol{G}$ があり

$$O\bigstar x = x\bigstar O = x$$

である. この O は単位元である.

群の性質 4. x に対して $\tilde{x} :: \boldsymbol{G}$ があり

$$\tilde{x}\bigstar x = O$$

である. この $\tilde{x}$ は x の逆元である.

であった. これらの条件を少し緩め, 逆元が存在しなくても良い「緩やかな群」を考えてみる. この「緩やかな群」のことを**単位的半群**または**モノイド**と呼ぶ.

モノイドは群に制約条件をひとつ付け加えたものとも言えるので, あらゆる群は自動的にモノイドでもある. プログラマの言葉で言えば, モノイドは群の親クラスである.

モノイドの性質は次の三つである. ただし x,y,z が集合 $\boldsymbol{M}$ の元であるとする.

モノイドの性質 1. x と y の演算の結果 $x\bigstar y$ は $\boldsymbol{M}$ の元である.

モノイドの性質 2. x,y,z について

$$(x\bigstar y)\bigstar z = x\bigstar(y\bigstar z)$$

である. これを結合則と呼ぶ.

モノイドの性質 3. $O :: \boldsymbol{M}$ があり

$$O\bigstar x = x\bigstar O = x$$

である. この O は単位元である.

—

$$+ :: \boldsymbol{Z} \mapsto \boldsymbol{Z} \mapsto \boldsymbol{Z}$$

整数全体の**集合**を $\boldsymbol{Z}$ で表すこととしよう. 集合 $\boldsymbol{Z}$ には 0 や -10 や 123 などのありとあらゆる整数が含まれている. 任意の変数 x が整数であることを, 数学者は $x \in \boldsymbol{Z}$ という風に書き, x が集合 $\boldsymbol{Z}$ の**元**であると呼ぶ. 我々は記号 $\in$ を別の用途に使いたいので代わりに記号 $::$ を使って $x :: \boldsymbol{Z}$ と書くことにする.

二つの整数

$$x, y :: \boldsymbol{Z}$$

の間には**二項演算**が定義されている. 例えば足し算は二項演算の例で, それを表現するために**演算子** $+$ がある. 整数 x,y に足し算演算子を**作用させる**と, つまり $x+y$ を作ると, その結果は再び整数になる. これを

$$x + y :: \boldsymbol{Z}$$

と書く. いま「演算子」を「作用」させたのだが, これはそれぞれ英語の “operator” と “operate” の訳である. 数学者や物理学者は演算子のことを**作用素**と呼ぶ場合もあるが, 意味は同じである.

演算子 $+$ は演算の順序によらず結果が同じである. いま三つの整数

$$x, y, z :: \boldsymbol{Z}$$

があるとき

$$x + (y + z) = (x + y) + z$$

であるが, このように右を先に計算しても左を先に計算しても結果が変わらないことを**結合性**と呼ぶ. また, 結合性があることを求める規則を**結合則**と呼ぶ. 整数の足し算は結合則に従う. 足し算にはもう一つ $x+y = y+x$ という性質があり, これを可換性と呼ぶが, 今は考えなくて良い.

整数の足し算にはもう一つの際立った特徴がある. それは $0, x :: \boldsymbol{Z}$ のとき

$$0 + x = x + 0 = x$$

であるような

$$0 :: \boldsymbol{Z}$$

が存在することだ. このように足し算を行っても結果を変えなような元のことを**単位元**と呼ぶ. 特に足し算の場合は単位元のことを**零元**と呼ぶこともある.

さて, これらの性質を一般化してみよう. 集合 $\boldsymbol{M}$ の元 $x,y,z :: \boldsymbol{M}$ とそれらの間の二項演算子 $\bigstar$ が,

$$x\bigstar y :: \boldsymbol{M} \tag{2.2}$$

$$x\bigstar(y\bigstar z) = (x\bigstar y)\bigstar z \ldots \text{結合則} \tag{2.3}$$

$$^{\exists}\mathcal{O}\,\text{where}\,\mathcal{O}\bigstar x = x\bigstar\mathcal{O} = x \ldots \text{単位元の存在} \tag{2.4}$$

を満たしているとき，組み合わせ $(\boldsymbol{M}, \bigstar, \mathcal{O})$ を**モノイド**と呼ぶ．

先ほど見たように整数の集合 $\boldsymbol{Z}$ と足し算演算子 $+$ と整数の 0 の組み合わせ $(\boldsymbol{Z}, +, 0)$ はモノイドである．整数については，掛け算演算子 $*$ に関しても $(\boldsymbol{Z}, *, 1)$ がモノイドである．

集合 $\boldsymbol{M}$ の元 $x, y, z \in \boldsymbol{M}$ とそれらの間の二項関係 $\bigstar$ が，

$$x \bigstar y :: \boldsymbol{M}$$
$$x \bigstar (y \bigstar z) = (x \bigstar y) \bigstar z \ldots \text{結合則}$$
$$^{\exists}\mathcal{O} \text{ where } \mathcal{O} \bigstar x = x \bigstar \mathcal{O} = x \ldots \text{単位元の存在}$$

を満たしているとき，組み合わせ $(\boldsymbol{M}, \bigstar, \mathcal{O})$ をモノイドと呼ぶ．

数学上の多くの構造がモノイドである．整数や二次元の回転変換がそうであったが，一般に数と呼ばれるもの全て，また正方行列も行列の掛け算（または足し算），単位行列（または零行列）とともにモノイドになる．ベクトルもまた，ベクトルの合成演算子，零ベクトルと組み合わせてモノイドになる．

そこで，今度は逆にモノイドでないものを考えてみよう．例えば三次元ベクトルのベクトル積はどうだろうか．三次元ベクトルのベクトル積は結合則が成り立たず，また単位元もないため，三次元ベクトル全体の集合とベクトル積はモノイドになることができない．

前節では二次元の回転変換がモノイドであることを示した．回転変換，すなわち回転の演算子と，回転される方のベクトルはそれぞれ異なる集合に属している．すべての回転変換の集合を $\boldsymbol{U}$，すべてのベクトルの集合を $\boldsymbol{V}$ とすると

$$\boldsymbol{U} :: \boldsymbol{V} \mapsto \boldsymbol{V}$$

となる．

前述のモノイドにもう一つ規則を加えたものが群である．群は集合 $\boldsymbol{G}$ の元 $x, y, z \in \boldsymbol{G}$ とそれらの間の二項関係 $\bigstar$ について

$$x \bigstar y :: \boldsymbol{G} \tag{2.5}$$
$$x \bigstar (y \bigstar z) = (x \bigstar y) \bigstar z \ldots \text{結合則} \tag{2.6}$$
$$^{\exists}\mathcal{O} : \mathcal{O} \bigstar x = x \bigstar \mathcal{O} = x \ldots \text{単位元の存在} \tag{2.7}$$
$$^{\exists}x^{-1} : x^{-1} \bigstar x = \mathcal{O} \ldots \text{逆元の存在} \tag{2.8}$$

であるような $(\boldsymbol{G}, \bigstar, \mathcal{O})$ のことであり，モノイドの規則に「逆元の存在」を加えたものである．プログラマの言葉で言えば，群はモノイドの「子クラス」であると言える．

前節で紹介した $(\boldsymbol{Z}, +, 0), (\boldsymbol{Z}, *, 1), (\boldsymbol{U}, \bullet, U_0)$ は全て群でもある．

このように規則を少しずつ増やしていって，より強力な概念を構築していく方法は数学ではよく用いられる．本書でもシンプルな規則から初めて，少しずつ規則を増やしていくやり方を多用する．

余計な話

第 3 章

記法

本書では一般の数学書やプログラミングの教科書からは少し異なった記法を用いる．ある概念が発明されてからずっと後になって正しい記法が見つかり，それがきっかけとなって正しく理解されるという現象は歴史上よくあることである．本書でも様々な新しい記号，記法を導入するが，この章では既に比較的広く知られている記法を紹介する．

3.1 関数

関数引数には括弧を付けない．我々はよく引数 x をとる関数 f を $f(x)$ と書くが，括弧は冗長なので今後は fx と書くことにする．引数 x を関数 f に「食わせる」ことを**関数適用**と呼ぶ．

複数引数をとる関数を我々はよく $f'(x,y)$ と書くが，これも括弧が冗長なので今後は $f'xy$ と書くことにする．この場合式 $f'xy$ は左を優先して結合するものとする．つまり

$$f'xy = (f'x)y$$

である．引数に「飢えた」関数 (fx) を**部分適用**された関数と呼ぶ．[*1]

このように式の左側を優先的に演算していくことを**左結合**と呼ぶ．関数適用は左結合である．他に式の右側を優先的に演算する**右結合**の演算子も存在する．

3.2 演算子

単項演算子には論理否定 $(\neg)$ とマイナス $(-)$ がある．本書で扱う基本的な単項演算子はこの 2 種類だけであるが，複数の演算子を組み合わせて単項演算子を新たに作ることはある．

二項演算子のうちよく使われるものは和 $(+)$，積 $(*)$，差 $(-)$，商 $(/)$，論理和 $(\vee)$，論理積 $(\wedge)$，同値 $(\equiv)$，大なり $(>)$，小なり $(<)$ 等である．二項演算子はたとえ積記号であっても省略できない．二項演算子は多数あるので，その都度説明する．

二項演算子は中置することが基本であるが，括弧で包むことで前置することも可能である．任意の二項演算子 ★ について

$$x★y$$

及び

$$(★)xy$$

は全く同じ意味である．[*2]

二項演算子の結合性，すなわち左結合か右結合かは，演算子によって異なる．

演算の優先順位を明示的に与えるために括弧が用いられる．

> 任意の二項演算子 ★ について
>
> $$x★y = (★)xy$$
>
> である．

3.3 ラムダ式

引数 x をとり値 $1+x$ を返す**ラムダ式**は次のように書くことにする．

$$\backslash x \to 1 + x$$

この式はラムダ式を発明したチャーチのオリジナルの論文の記法であれば $\hat{x}.1+x$ と書かれたところであり，現在でも多くの書物で $\lambda x.1+x$ と記述されるところである．しかし我々はすべてのギリシア文字を変数名のため

[*1] Haskell では関数 `f` に引数 `x` を適用させることを `f x` と書く．

[*2] Haskell では任意の二項演算子を括弧で包むことで前置演算子として使うことができる．例えば `x+y` と `(+)x y` は同じ結果を返す．逆に任意の二引数関数 `f` は `x `f` y` と書くことで中置することができる．

に予約しておきたいのと，ピリオド記号 (.) が今後登場する二項演算子 · と紛らわしいため，上述の記法を用いる．[*3]

ラムダ式は関数である．ラムダ式を適用するには，ラムダ式を括弧で包む必要がある．例を挙げる．

$$(\backslash x \to 1 + x)\, 2$$

この式は結果として 3 を返す．

複数引数をとるラムダ式は例えば

$$\backslash xy \to x + y$$

のように引数を並べて書く．

本書では新たに，次のラムダ式記法も導入する．式中に記号 ◊ が現れた場合，その式全体がラムダ式であるとみなす．記号 ◊ の部分には引数が入る．第 n 番目の ◊ には第 n 番目の引数が入る．例えばラムダ式 $\backslash xy \to x + y$ は

$$◊ + ◊$$

と書いても良い．式を左から読んで 1 番目の ◊ が元々の x すなわち第 1 引数を，2 番目の ◊ が元々の y すなわち第 2 引数を意味する．この省略記法はプログラミング言語 Scheme における cut プロシジャに由来する．[*4]

[*3] Haskell ではこのラムダ式を `\x->1+x` と書く．

[*4] この記法は Haskell にはない．

第 4 章

関数

プログラムはひとつの関数であり，その関数はいくつかの部分的な関数から合成される．つまり，関数はプログラムの全体であり，またビルディングブロックである．プログラムという関数をより小さな関数の合成で作るのは，それらの小さな関数が他のプログラムに再利用されることを期待するためである．このように関数と関数の合成はプログラミングの中心的な概念である．

4.1 関数の定義

ラムダ式を用いた**関数**の定義が可能である．例えば引数 x をとり値 $1+x$ を返す関数 f は

$$f = \backslash x \to 1 + x$$

と定義できる．この省略形として

$$fx = 1 + x$$

と書いても良いし，さらに省略して

$$f = (+)1$$

と書いても良い．もちろん $f = +1$ でないことに注意しよう．[*1]

関数に**スペシャルバージョン**がある場合はそれらを列挙する．例えば引数が 0 の場合は特別に戻り値も 0 であり，その他の場合は関数 f と同じ振る舞いをする関数 f' を考える．このとき f' は

$$f'0 = 0$$
$$f'x = 1 + x$$

ように定義することになる．[*2]

関数定義に**場合分け**が必要な場合は「ガード」を用いる．例えば引数の値が負の場合は -1 を，0 の場合は 0 を，それ以外の場合は関数 f と同じ振る舞いをする関数 f'' は

$$f''x \,|_{x<0} = -1$$
$$\quad |_{x\equiv 0} = 0$$
$$\quad |_{\mathbf{otherwise}} = 1 + x$$

という風に定義することにする．[*3]

4.2 関数の合成

関数は**合成**できる．関数 f と関数 g があって，その合成を $f \bullet g$ と書くとき

$$(f \bullet g)x = f(gx)$$

である．関数合成の演算子 $\bullet$ は関数適用よりも優先順位が高く，$(f \bullet g)x$ は単に $f \bullet g\, x$ と書いても良い．この記法は括弧の数を減らすためにしばしば用いられる．[*4]

関数合成演算子とは逆に，優先順位の低い関数適用演算子も考えておくと括弧の数を減らすのに便利である．関数適用演算子 $ を次のように定義しておく．

$$f\,\$\,gx = f(gx)$$

演算子 $ の優先順位は足し算演算子よりも低いものとする．よって $f(x+1)$ は $f\,\$\,x+1$ と書くこともできる．[*5]

[*1] Haskell では $f = \backslash x \to 1 + x$ を `f=\x->1+x`，$fx = 1 + x$ を `f x = 1+x`，$f = (+)1$ を `f=(+)1` のように書く．

[*2] Haskell では

```
f' 0 = 0
f' x = 1+x
```

と書く．

[*3] Haskell では

```
f'' x | x<0       = -1
      | x==0      = 0
      | otherwise = 1+x
```

と書く．

[*4] Haskell では関数 `f` と関数 `g` の合成は `f.g` である．

[*5] Haskell では $f\,\$\,gx$ を `f $ g x` と書く．

4.3　条件式

条件式とは場合分けを関数定義の右辺に書けるようにしたもので，

$$f x = \mathbf{if}\, x \equiv 0\, \mathbf{then}\, 0\, \mathbf{else}\, 1 + x$$

のように **if** 節，**then** 節，及び **else** 節からなるものである．**if** 節の中身は真理値を返す関数であれば良いので，関数 p を

$$p = (\equiv)0$$

としておき，

$$f x = \mathbf{if}\, p x\, \mathbf{then}\, 0\, \mathbf{else}\, 1 + x$$

と書く方法もしばしば用いられる．[*6]

[*6] Haskell では $f x = \mathbf{if}\, x \equiv 0\, \mathbf{then}\, 0\, \mathbf{else}\, 1 + x$ を `f x = if x == 0 then 0 else 1+x` と書く．

第 5 章

型と型クラス

プログラマのいう型とは，数学者のいう代数構造のことである．

5.1 型

型とは変数が取りうる値に与えた制約のことである．数学者がよく扱う型は論理型 ($\boldsymbol{B}$)，整数型 ($\boldsymbol{Z}$)，実数型 ($\boldsymbol{R}$) といったところであろう．ただし数学者たちは型ではなく集合と呼ぶ．括弧内に示した記号は数学者たちが慣用的に用いているものである．

我々は計算機での実装を考慮して，数学者たちとは若干異なる型と表記方法を扱うことにする．まず論理型 ($\boldsymbol{B}$) はそのまま論理型として **Bool** で表す．

計算機にとって都合の良い整数の範囲を考慮した整数型を **Int** で表す．計算機にとって都合の良い整数の範囲とは，例えば 64 ビット計算機の場合 -2^{32} から $2^{32}-1$ までの範囲のことである．

計算機は残念ながら無限精度の実数を扱えない．そこで標準精度の浮動小数点数を表す型を **Float** で表す．

もう一つ，計算機ならではの型を導入しておこう．それは **Int** とよく似ているが，特別に文字を扱うために考えられた型 **Char** である．文字といってもその中身は整数である．整数ではあるが，わざわざ別な型とするのには理由がある．歴史的には文字は小さな整数として表現されることが多かったため，また計算機のメモリが高価であったため，文字を扱うための専用の型はもっぱらメモリの節約のために存在した．現在では，文字が数値にエンコードされる方式がより複雑になってきたために，文字を通常の数値と区別するために特別な型が用意さてれいるのである．

表 5.1　モノイド

型クラス	$\mathfrak{Monoid}$	
型 $\setminus$ 要素と演算子	$\mathcal{O}$	$\bigstar$
Char	—	—
Bool	F	$\vee$
Bool	T	$\wedge$
Int	0	$+$
Int	1	$*$
Float	0	$+$
Float	1	$*$

5.2 関数の型

関数にも型がある．例えば整数引数を一つ取り，整数を返す関数 f は

$$f :: \mathbf{Int} \mapsto \mathbf{Int}$$

という型を持つ．

整数引数を二つ取り，整数を返す関数 f' は

$$f' :: \mathbf{Int} \mapsto \mathbf{Int} \mapsto \mathbf{Int}$$

という型を持つ．これは

$$f' :: \mathbf{Int} \mapsto (\mathbf{Int} \mapsto \mathbf{Int})$$

と同じ意味である．

5.3 型クラス

$(\mathbf{Int}, +, 0)$ はモノイドである．同様に $(\mathbf{Int}, *, 1)$, $(\mathbf{Float}, +, 0)$, $(\mathbf{Float}, *, 1)$, $(\mathbf{Bool}, \vee, \mathrm{F})$, $(\mathbf{Bool}, \wedge, \mathrm{T})$ もモノイドである．そこで，任意の型 $\mathbf{a}$ について，組み合わせ $(\mathbf{a}, \bigstar, \mathcal{O})$ がモノイドである場合には

$$(\mathbf{a}, \bigstar, \mathcal{O}) \subset \mathfrak{Monoid}$$

と書くことにする．二項演算子，単位元が自明な場合は簡略化して

$$\mathbf{a} \subset \mathfrak{Monoid}$$

と書くことにする．

このような型をより抽象化した型のようなものを型クラスと呼ぶ．モノイドは型クラスの例である．表 5.1 に型と対応するモノイドの単位元，演算子の一覧を示す．

もっと身近な型クラスもある．例えば，型 **a** の変数どうしの間で等値性が定義されている場合，その型は型クラス $\mathfrak{Eq}$ に属することになる．型クラス $\mathfrak{Eq}$ に属する型は等値演算子 $\equiv$ を提供する．

型 **a** の変数どうしの間で大小関係が定義されている場合，かつその型が型クラス $\mathfrak{Eq}$ に属する場合，その型は型クラス $\mathfrak{Ord}$ にも属する．型クラス $\mathfrak{Ord}$ に属する型は比較演算子 $<, \leq, \geq, >$ を提供する．

型 **a** の変数どうしの間で四則演算関係が定義されている場合，かつその型が型クラス $\mathfrak{Eq}$ に属する場合，その型は型クラス $\mathfrak{Num}$ にも属する．型クラス $\mathfrak{Num}$ に属する型は比較演算子 $+, -, *, /$ を提供する．

型 **a** が型クラス $\mathfrak{Ord}$ 及び型クラス $\mathfrak{Num}$ に属しているとき，かつそのときに限り，型 **a** は型クラス $\mathfrak{Real}$ にも属する．

型 **a** の変数について，一つ小さい値を返す関数 pred と一つ大きい値を返す関数 succ が定義されているとき，かつそのときに限り，型 **a** は型クラス $\mathfrak{Enum}$ に属する．

型 **a** が型クラス $\mathfrak{Real}$ 及び型クラス $\mathfrak{Enum}$ に属しているとき，かつそのときに限り，型 **a** は型クラス $\mathfrak{Integral}$ にも属する．

これらの関係を表にまとめたものが表 5.2 である．

表 5.2 型と型クラス

型クラス	Integral											
	Real										Enum	
	Ord					Num						
	Eq					Eq						
型 \ 演算子	$\equiv$	$<$	$\leq$	$\geq$	$>$	$\equiv$	$+$	$-$	$*$	$/$	pred	succ
Char	✓	✓	✓	✓	✓	✓	—	—	—	—	✓	✓
Bool	✓	✓	✓	✓	✓	✓	—	—	—	—	✓	✓
Int	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Float	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	—	—

第 6 章

コンテナ

型から作る型をコンテナと呼ぶ．代表的なコンテナはある型のホモジニアスな配列であるリストである．この章ではリストと，リストに対する重要な演算である畳み込み，マップを取り扱う．

6.1 リスト

同じ型の値を一列に並べたもの，つまりホモジニアスな配列のことを**リスト**と呼ぶ．例えば 0 から始まり 9 まで続く整数のリストは $[0,1,2,3,4,5,6,7,8,9]$ と書く．等差数列に限って，簡略化した書き方が許される．0 から 9 までのリストは $[0,1\ldots9]$ と書いても良い．複数の型の要素が混在してもよい配列のことをヘテロジニアスな配列と呼び，ホモジニアスな配列とは区別する．

より高度なリスト，例えば 0 から 9 までの平方数のリストは

$$\left[x^2 \,|_{x\in[0,1\ldots9]}\right]$$

のようにガードを用いて書く．ここに右辺のリストから一つずつ要素を取り出して左辺に代入する演算子 $\in$ を用いた．

ガードの中の式は複数あっても良い．例えば

$$\left[x+y \,|_{x\in[0,1\ldots9],\, y\in[0,1\ldots5],\, x+y>3}\right]$$

は $0\leq x\leq 10$ かつ $0\leq y\leq 5$ の範囲で $x+y>3$ となる x 及び y から $x+y$ を並べたリストである．

リストは**結合**できる．例えばリスト l とリスト m を結合したリストは

$$l\oplus m$$

で得られる．

リストは無限個の要素を持っても良い．例えば自然数全体を表すリスト n は

$$n=[1,2\ldots]$$

のように定義して良い．

空リストは $\emptyset$ で表す．

任意の型を **a** とするとき，**a** 型のリストを **List a** と書く．**List** のように型名 **a** を修飾するものを**型コンストラクタ**と呼ぶ．

ある型を包み込んだ別の型を一般に**コンテナ型**または単に**コンテナ**と呼ぶ．

6.2 畳み込み

我々はよくリストの総和を表現するためにシグマ記号 $(\sum)$ を使う．リスト $[x_0,x_1\ldots x_n]$ の総和を

$$\sum[x_0,x_1\ldots x_n]=x_0+x_1+\cdots+x_n$$

とするようなシグマ記号である．この表現を一般化してみよう．リスト $[x_0,x_1\ldots x_n]$ が与えられたとき，

$$\bigcup_a^{\bigstar}[x_0,x_1\ldots x_n]=a\bigstar x_0\bigstar x_1\bigstar\cdots\bigstar x_n$$

であると定義する．この新しい記号 $\bigcup$ は**畳み込み演算子**と呼ばれる．変数 a はアキュムレータと呼ぶ．アキュムレータは右側の引数が空であった場合のデフォルト値と考えても良い．

リストの総和をとる演算子 $\sum$ は

$$\sum=\bigcup_0^+$$

とすれば得られる．同じようにリストの要素のすべての積をとる演算子 $\prod$ は

$$\prod=\bigcup_1^*$$

とすれば得られる．

畳み込み演算子は第 1（上）引数に **a** 型と **b** 型の引数を取り **a** 型の戻り値を返す二項演算子，第 2（下）引数

に **a** 型，第 3（右）引数に **b** 型のリストすなわち [**b**] 型を取り，**a** 型の値を返す．従って畳み込み演算子の型は

$$\bigcup :: (\mathbf{a} \mapsto \mathbf{b} \mapsto \mathbf{a}) \mapsto \mathbf{a} \mapsto [\mathbf{b}] \mapsto \mathbf{a}$$

と定義できる．

畳み込み演算子には次のようなもう一つのバリエーションがある．

$$\bigsqcup_{a}^{\bigstar} [x_0, x_1 \dots x_n] = (x_0 \bigstar (x_1 \bigstar \cdots \bigstar (x_n \bigstar a)))$$

これは右畳み込みと呼ばれる演算子である．

畳み込み演算子の面白い応用例を示そう．

$$\bigcup_{\emptyset}^{\oplus} [[0,1,2],[3,4,5],\dots] = [0,1,2,3,4,5,\dots]$$

であるから，演算子

$$\bigcup_{\emptyset}^{\oplus}$$

はリストを平坦化する**平坦化演算子**である．平坦化演算子は concat 演算子とも呼ばれることもあるが，基本的な演算子であるため特別な記号をつけておこう．我々は

$$\dagger = \bigcup_{\emptyset}^{\oplus}$$

と定義することにする．

6.3 マップ

リストの各要素に決まった関数を適用したい場合がある．引数として関数 f とリスト $[x_0, x_1 \dots x_n]$ を取り，戻り値として $[fx_0, fx_1 \dots fx_n]$ を返す演算子 $\odot$ を考えよう．このとき

$$f \odot [x_0, x_1 \dots x_n] = [fx_0, fx_1 \dots fx_n]$$

であると定義する．

第 7 章

コンテナの高度な話題

a

7.1 Maybe

整数 x を 0 で除算することはエラーである．しかし除算の度にエラーが起こったかどうか調べるのは面倒であるし，記述も煩雑になってしまう．プログラマはよくこういう場合に「例外」という機構を用いるが，数学に例外は持ち込みたくない．そこで変数 x が正しく計算されたかもしれないし，されていないかもしれないということを，特別な記号 $\{x\}$ で表しておこう．ここで変数 $\{x\}$ が取り得る値は正しく計算された値 x か，あるいはエラーを表す値 $\varnothing$ である．

この変数 $\{x\}$ はもはや整数型とは言えない．そこでこの $\{x\}$ の型を **Maybe Int** と表して「きっと整数」型と呼ぶことにしよう．ここに **Maybe** は型コンストラクタのひとつである．

変数 x が一度 0 除算の危険性に汚染された場合，その後ずっと $\{x\}$ と印をつけられなければならない．このきっと整数 $\{x\}$ に対して整数を引数にとる関数 f を適用させるには，何らかの関数マップ演算子が必要である．そこできっと整数の中身に直接関数を適用する演算子を $\boxdot$ としよう．具体的には

$$f \boxdot \{x\} = \{fx\}$$

のように関数 f を変数 $\{x\}$ に適用させる．その結果はまた **Maybe Int** 型である．従って，結果を受け取る変数も **Maybe Int** 型でなければならず，それを忘れないように印を付けておかねばならない．例えば

$$\{y\} = f \boxdot \{x\}$$

のように書くことになるであろう．

$$(\mathbf{Maybe\ a}, \boxplus, \varnothing)$$

7.2 モノイドとしてのコンテナ

$$(\mathbf{List\ a}, \oplus, \emptyset)$$

7.3 リストの実装

ここでリストの実装について若干述べておかねばならない．紙上ではリストは自由に考えられるが，計算機上ではそれほど自由ではないからである．ここではリストは LISP におけるリストと同じ構造を持つものとする．LISP におけるリストとは要素 First と要素 Rest からなるペアの集合である．要素 First がリストの要素を持ち，要素 Rest が次のペアを参照する．リストの最後のペアの Rest は空リストを参照する．

リストのための特別な表現

$$\mathsf{First} : \mathsf{Rest}$$

を用い，第 1 要素 First はリストが保持する型，第 2 要素 Rest はリスト型であるとする．演算子 : を**結合演算子**と呼ぶ．

要素 Rest はリストまたは空リストであるから，一般にリストは次のように展開できることになる．

$$\begin{aligned}[x_0, x_1, x_2 \ldots x_n] &= x_0 : [x_1, x_2 \ldots x_n] \\ &= x_0 : x_1 : [x_2 \ldots x_n] \\ &= x_0 : x_1 : x_2 : \cdots : x_n : \emptyset\end{aligned}$$

結合演算子 : は右結合する．すなわち

$$x_0 : x_1 : x_2 = x_0 : (x_1 : x_2)$$

である．

第 8 章

関手

8.1 関手

$$F :: \boldsymbol{X} \mapsto \boldsymbol{X}$$

$$\mathcal{I}\cdot = \mathcal{I} \tag{8.1}$$
$$(f \bullet g)\cdot = (f\cdot) \bullet (g\cdot) \tag{8.2}$$

—

ある集合 $\boldsymbol{S}$ と，その集合の元に対して定義される変換 f 全体の集合 $\boldsymbol{F_S}$ との組み合わせ $(\boldsymbol{S}, \boldsymbol{F_S})$ を**圏**と呼ぶ．例えばリストは $(\mathbf{List\ a}, \odot)$ という圏であるし，Maybe は $(\mathbf{Maybe\ a}, \boxdot)$ という圏である．

これらの圏はもともと $(\mathbf{a}, \bigstar)$ という圏から変換して得られたものと考えることもできる．つまり，

$$\mathbf{List} = (\mathbf{a}, \bigstar) \mapsto (\mathbf{List\ a}, \odot)$$
$$\mathbf{Maybe} = (\mathbf{a}, \bigstar) \mapsto (\mathbf{Maybe\ a}, \boxdot)$$

であると考えることができる．このような圏から圏への変換を**関手**と呼ぶ．

一般に

$$\mathbb{F} = (\boldsymbol{a}, \bigstar) \mapsto (\mathbb{F}\ \mathbf{a}, \cdot)$$

—

モノイド $(\boldsymbol{M}_1, \bigstar_1, \mathcal{O}_1)$ と $(\boldsymbol{M}_2, \bigstar_2, \mathcal{O}_2)$ があるとき，前者から後者への変換 $\mathbb{F}$ を

$$\mathbb{F} = (\boldsymbol{M}_1, \bigstar_1, \mathcal{O}_1) \mapsto (\boldsymbol{M}_2, \bigstar_2, \mathcal{O}_2)$$

と書き，変換 $\mathbb{F}$ を**関手**と言う．

例えば **a** 型から **List a** 型を作る型コンストラクタ **List** は関手である．この場合は

$$\mathbf{List} = (\mathbf{a}, +, 0) \mapsto (\mathbf{List\ a}, \odot, \emptyset)$$

という関係になる．

また **a** 型から **Maybe a** 型を作る型コンストラクタ **Maybe** も関手である．この場合は

$$\mathbf{Maybe} = (\mathbf{a}, +, 0) \mapsto (\mathbf{Maybe\ a}, \boxplus, \varnothing)$$

という関係になる．

モノイドからモノイドへの変換，モノイド内での二項演算子なども含めて，変換するものを全て**射**と呼ぶ．関手は射のひとつである．

8.2 アプリカティブ関手

$$(X, \langle\rangle, \cdot, \times)$$

8.3 関数と関手の関係

第 9 章

モナド

9.1 アプリカティブ関手の拡張

$$(X, \langle\rangle, \cdot, \times, \heartsuit)$$

9.2 モナド則

9.3 モノイドなモナド

型クラス	Monadplus					
	Monad				Monoid	
	Applicative					
	Functor					
型 \ 要素と演算子	$\cdot$	$\langle x \rangle$	$\times$	$\heartsuit$	$\mathcal{O}$	$\bigstar$
リスト	$\odot$	$[x]$	$\otimes$	$\circ$	$\emptyset$	$\oplus$
Maybe	$\boxdot$	$\{x\}$	$\boxtimes$	$\square$	$\varnothing$	$\boxplus$
関数	$\bullet$	$[\![x]\!]$	$\bowtie$	$\diamond$	$\lozenge$	$\bullet$
Int（参考）					0	$+$

第 II 部

モナドから見る世界

第 10 章

IO

第 11 章

例外

第 12 章

乱数

第 13 章

状態

第 14 章

非決定性

第 15 章

継続

第 16 章

X

第 17 章

Y

第 18 章

Z

第 III 部

圏とモナド

第 19 章

—

第 20 章

思考の道具*

「言語は思考を規定する」という．実際プログラマは思考の道具としてプログラミング言語をよく用いるし，そのプログラマの好みのプログラミング言語の影響を強く受ける．手続き的に問題を解くのが好きなプログラマもいれば，宣言的に問題を解くのが好きなプログラマもいる．問題の複雑さを「オブジェクト」というプログラム単位に押し込むのが好きなプログラマもいれば，「クロージャ」という別のプログラム単位に押し込むのが好きなプログラマもいる．

一方で，古代ギリシャから連綿と続く数学者たちは，数学という言葉で問題を考えることを好む．数学はわずかな方言の違いを無視すれば驚くほど統一された言語である．

プログラマが使う言葉すなわちプログラミング言語と，数学者たちが使う語はだいたい乖離しており，その両者の歩み寄りは虚しいものであることが多かった．ここで言いたいのはプログラマが限られた文字セットしか使えないとか，スクリーン上の行という制約に縛られているとか，そのような表面的なことではない．そうではなく，思考の様式が，プログラミング「文化」と数学「文化」で異なるという意味である．

プログラミング文化と数学文化の大きな違いの一つは，リアルタイムに起こるイベントの取り扱い方法である．プログラマはほとんどいつもリアルタイムに起こるイベントに対応しないといけない．一方で，数学者たちはそのようなイベントを「不純なもの」として理論から取り除く．例えば変数 x があるときは 0 だがあるときは 1 であるというのは，甚だ不純なものとして数学者の間では認識されるものだ．

第 21 章

数学的準備*

この章ではいま後本書で登場する数学的概念の記法を決めておく．

21.1 定数・論理定数・文字定数

定数はアラビア数字で記述するものとする．定数が負の場合はマイナス記号 (−) を用いる．定数が実数の場合は小数点 (.) を用いる場合がある．例えば $-1, 0, 1.2$ は定数である．

論理定数とは真偽値を表す定数で，真 (T) 及び偽 (F) の 2 種類のみがある．

文字定数とはコード化された文字のことである．文字コードを定数として扱うといささか読みづらくなるため，該当する文字をシングルクォートで囲むことにする．例えば ‘A’, ‘b’, ‘.’ は文字定数である．

21.2 変数

変数はアルファベット（ローマ文字またはギリシア文字）の小文字 1 文字で表すことにする．変数名には任意個のプライム記号 (′) をつけても良いし，添え字をつけても良いこととする．変数名はイタリック体で書く．

変数は代入演算子 (=) によって定数や変数，それらを複合した式の値を代入できる．例えば

$$x = 1, y = x$$

は代入の例である．

21.3 演算子

演算子には数学記号を割り当てる．

21.4 関数

関数も変数と同じようにアルファベット（ローマ文字またはギリシア文字）の小文字 1 文字で表すことにする．関数名には任意個のプライム記号 (′) をつけても良いし，添え字をつけても良いこととする．三角関数のように十分な知名度を持つ関数や，それに匹敵する汎用的な関数は長い名前を持っても良い．変数名や関数名はイタリック体で書くが，長い関数名はローマン体で書く．

二項演算子とは 2 引数関数の特別な場合であると考えてよい．関数 r を次のように中置する記法を定義しておく．

$$x \grave{} r \grave{} y \xleftarrow{\text{def}} rxy$$

バックプライム記号 (‵) を使っているのはプライム記号 (′) と区別するためである．

既存の二項演算子 $\circ$ は次のようにして通常の関数として使えるものとする．

$$(\circ)xy \xleftarrow{\text{def}} x \circ y$$

21.5 ラムダ式

21.6 関数定義

21.7 条件式

21.8 型

21.9 リスト

21.10 タプル

複数の変数を束ねたものをタプルと呼ぶ．変数 x と変数 y からなるタプルは (x, y) と書く．いま

$$t = (x, y)$$

としたとき，タプルの中身を取り出すには

$$(t_1, t_2) = t$$

のようにしてパタンマッチングを用いる．

21.11 リストの実装

第 22 章

型・型クラス・種*

22.1 関数の型

$$f :: \mathbf{Z} \to \mathbf{Z}$$

関数 f が引数として整数を 2 個取り 1 個の整数を返すとき，その型は

$$\mathbf{Z} \to \mathbf{Z} \to \mathbf{Z}$$

である．型の式は右結合するものとする．従って上式は

$$\mathbf{Z} \to (\mathbf{Z} \to \mathbf{Z})$$

と読む．これは関数 f が最初の引数を 1 個受け取り，$(\mathbf{Z} \to \mathbf{Z})$ 型の関数を返していることを考えると理にかなっている表現である．

22.2 型クラス

$$\mathbf{Z} \subset \mathfrak{Q}$$

$$\mathbf{Z} \subset \mathfrak{O}$$

$$\mathbf{Z} \subset \mathfrak{E}$$

22.3 種

$$\mathbf{Z} :: *$$

$$\{\mathbf{Z}\} :: *$$

$$\{\} :: * \mapsto *$$

第 23 章

マップ*

Map.

$$f \odot [x_0, x_1 \cdots x_n] = [fx_0, fx_1 \cdots fx_n]$$
$$f \odot \{x\} = \{fx\}$$

Applicative map (fmap).

$$[f_0, f_1 \cdots f_n] \otimes [x_0, x_1 \cdots x_{n'}] = [f_0x_0, f_0x_1 \cdots f_0x_{n'}, f_1x_0, f_1x_1 \cdots]$$
$$\{f\} \otimes \{x\} = \{fx\}$$
$$f \otimes g = f \bullet g$$

Monadic map (bind).

$$f \oplus [\![x]\!] = [\![fx]\!]$$

23.1 畳み込み

23.2 右畳み込み

23.3 フィルタ

$$\bigcap^{p} [x_0, x_1 \cdots x_n] \xleftarrow{\text{def}} \left[x \,|_{x \in [x_0, x_1 \cdots x_n] \wedge px \equiv \mathrm{T}}\right]$$

23.4 マップ

23.5 一般化マップ

マップ演算子の概念を一般化してみよう．一般のマップ演算子はあるコンテナの中身に関数適用を行い，元のコンテナと同じコンテナを返す演算子であると考えることができる．そこで，いま後マップ演算子の第 2（右）引数は一般のコンテナであるとする．型 $\mathbf{a}$ をコンテナに入れた型を一般に $\langle\!\langle \mathbf{a} \rangle\!\rangle$ と書くことにすると，一般化されたマップ演算子 $\bigodot$ は

$$\bigodot :: (\mathbf{a} \to \mathbf{b}) \to \langle\!\langle \mathbf{a} \rangle\!\rangle \to \langle\!\langle \mathbf{b} \rangle\!\rangle$$

という型を持つ．

マップはリストのマップのより汎用的なバージョンである．つまり我々は $\bigodot_{\text{list}}$ を使わずに，最初から $\bigodot$ を使えば良いのである．

一般化マップの型は $\mathbf{a} \to \mathbf{b}$ をとり $\langle\!\langle \mathbf{a} \rangle\!\rangle \to \langle\!\langle \mathbf{b} \rangle\!\rangle$ を返しているようにも見える．それゆえ一般化マップを用いることを関数の「持ち上げ (lifting)」と呼ぶのである．

23.6 Maybe

23.7 関手

ある型から，マップの第 2（右）引数になれる型を作る射（一般化された写像）のことを関手と呼ぶ．例えば型 $\mathbf{a}$ からリスト型 $[\mathbf{a}]$ を作る射 **List** ただし

$$\mathbf{List}\,\mathbf{a} \xleftarrow{\text{def}} \mathbf{a} \mapsto [\mathbf{a}]$$

は関手である．また型 $\mathbf{a}$ から “maybe a” 型 $\{\mathbf{a}\}$ を作る射 **Maybe** ただし

$$\mathbf{Maybe}\,\mathbf{a} \xleftarrow{\text{def}} \mathbf{a} \mapsto \{\mathbf{a}\}$$

も関手である．

関数合成もまた関手の例である．いま型 $\mathbf{a}$ を型 $\langle \mathbf{a} \rangle$ に変換する関手

$$\mathbf{Function}\,\mathbf{a} \xleftarrow{\text{def}} \mathbf{a} \mapsto \langle \mathbf{a} \rangle$$

を考える．型 $\langle \mathbf{a} \rangle$ は

$$\langle \mathbf{a} \rangle \xleftarrow{\text{def}} \mathbf{t} \to \mathbf{a}$$

であるとする．ここに型 $\mathbf{t}$ は任意の型である．関数 f が

$$f :: \mathbf{a} \to \mathbf{b}$$

であり，関数 g が

$$g :: \mathbf{t} \to \mathbf{a}$$

であるとき，その合成 $f \bullet g$ は

$$f \bullet g :: \mathbf{t} \to \mathbf{b}$$

でなければならない．ここで型 $\langle \mathbf{a} \rangle$ の定義式を用いると

$$\langle \mathbf{a} \rangle \xleftarrow{\text{def}} \mathbf{t} \to \mathbf{a}$$
$$\langle \mathbf{b} \rangle \xleftarrow{\text{def}} \mathbf{t} \to \mathbf{b}$$

であるから，関数 g 並びに合成関数 $f \bullet g$ は

$$g :: \langle \mathbf{a} \rangle$$
$$f \bullet g :: \langle \mathbf{b} \rangle$$

という型を持つことがわかる．今一度マップ演算子の型

$$\bigodot :: (\mathbf{a} \to \mathbf{b}) \to \langle\!\langle \mathbf{a} \rangle\!\rangle \to \langle\!\langle \mathbf{b} \rangle\!\rangle$$

を思い出してみよう．これは関数合成にもそのまま使えることがわかる．実際

$$f \bullet g \xleftarrow{\text{def}} \overset{f}{\bigodot} g$$

として関数合成は定義できる．

一般に

$$\mathbf{a} \mapsto \langle\!\langle \mathbf{a} \rangle\!\rangle$$

という風に型 $\mathbf{a}$ をコンテナに入れて型 $\langle\!\langle \mathbf{a} \rangle\!\rangle$ を作る射を関手と呼ぶ．

関手全体の型クラスを $\mathfrak{F}$ で表す．これまでに見た関手は

$$\mathbf{Function}, \mathbf{List}, \mathbf{Maybe} :: \mathfrak{F}$$

である．

23.8 関手が従う規則

$$\overset{\mathcal{O}}{\bigodot} = \mathcal{O} \tag{23.1}$$

$$\overset{\mathbf{f} \bullet \mathbf{g}}{\bigodot} \equiv \overset{\mathbf{f}}{\bigodot} \bullet \overset{\mathbf{g}}{\bigodot} \tag{23.2}$$

23.9 アプリカティブ関手

一般化マップとは

$$\bigodot :: (\mathbf{a} \to \mathbf{b}) \to \langle\!\langle \mathbf{a} \rangle\!\rangle \to \langle\!\langle \mathbf{b} \rangle\!\rangle$$

のことであった．一般化マップの第 1 引数である 1 引数関数がコンテナに入っている場合を想定しよう．そうすると次のような「一般化された」マップ $\bigotimes$ を考えることができる．

$$\bigotimes :: \langle\!\langle (\mathbf{a} \to \mathbf{b}) \rangle\!\rangle \to \langle\!\langle \mathbf{a} \rangle\!\rangle \to \langle\!\langle \mathbf{b} \rangle\!\rangle$$

実際の使い方は

$$[y_0, y_1 \cdots y_{n''}] = \overset{[f_0, f_1 \cdots f_n]}{\bigotimes} [x_0, x_1 \cdots x_{n'}]$$

のようになるであろう．

一般化マップ演算子 $\bigotimes$ と次のピュア演算子 $\bigstar$ があれば，リストも Maybe も同じ枠組みで考えることができる．ピュア演算子 $\bigstar$ は

$$\bigstar :: \mathbf{a} \to \langle\!\langle \mathbf{a} \rangle\!\rangle$$

という型を持つ．関手の型 $\mathbf{a} \mapsto \langle\!\langle \mathbf{a} \rangle\!\rangle$ とは矢印が違う事に注意しよう．ピュア演算子は型を型へ変換するのではなく，変数をコンテナに入れられた別な変数へと変換する．

型 $\langle\!\langle \mathbf{a} \rangle\!\rangle$ の変数をアプリカティブと呼ぶ．

リストの場合，ピュア演算子と一般化マップは次のように実装することができる．

$$\bigstar x = [x]$$

$$\overset{[f_0, f_1 \cdots f_n]}{\bigotimes} [x_0, x_1 \cdots x_{n'}] = \left[f x \,|_{f \in [f_0, f_1 \cdots f_n], x \in [x_0, x_1 \cdots x_{n'}]} \right]$$

Maybe の場合は次のようにピュア演算子と一般化マップを実装することができる．

$$\bigstar x = \{x\}$$

$$\overset{\mathsf{Nothing}}{\bigotimes} \{x\} = \mathsf{Nothing}$$

$$\overset{\{f\}}{\bigotimes} \{x\} = \overset{f}{\bigodot} \{x\}$$

アプリカティブが従う規則

$$\overset{\bigstar f}{\bigotimes} x \equiv \overset{f}{\bigodot} x$$

23.10 モナド

ある型 $\mathbf{a}$ について，特殊なコンテナ $[\![\mathbf{a}]\!]$ に入れられた型を想像してほしい．この型にもマップ演算子 $\bigoplus$ があり

$$\bigoplus :: (\mathbf{a} \to [\![\mathbf{b}]\!]) \to [\![\mathbf{a}]\!] \to [\![\mathbf{b}]\!]$$

という型を持っているとする．このマップ演算子は第 1（上）引数として型 $\mathbf{a}$ をとりコンテナに収めた型 $[\![\mathbf{b}]\!]$ を返す関数を，第 2（右）引数としてコンテナに収められた型 $[\![\mathbf{a}]\!]$ をとり，戻り値としてコンテナに収められた型 $[\![\mathbf{b}]\!]$ を返す．

型 $\mathbf{a}$ から型 $[\![\mathbf{a}]\!]$ を作り出す関手を **Monad** とし

$$\mathbf{Monad}\,\mathbf{a} = \mathbf{a} \mapsto [\![\mathbf{a}]\!]$$

とする．$\mathbf{a}$ 型の変数から $[\![\mathbf{a}]\!]$ 型の変数を作る演算子を $\star$ とし，ユニット演算子と呼ぶ．

$[\![\mathbf{a}]\!]$ 型の変数をモナドと呼ぶ．モナドはアプリカティブとよく似ているが，少し異なることに注意しよう．モナドの応用範囲は極めて広く，それゆえ基本的な関手と考えられている．

Maybe の方がリストよりも簡単なので，Maybe から説明する．Maybe の場合は次のような実装となる．

$$\star x = \{x\}$$

$$\bigoplus^{f} \mathsf{Nothing} = \mathsf{Nothing}$$

$$\bigoplus^{f} \{x\} = fx$$

ここに関数 f は

$$f :: \mathbf{a} \to [\![\mathbf{b}]\!]$$

であるから，関数適用された値 fx は

$$fx :: [\![\mathbf{b}]\!]$$

という型を持つことに注意しよう．

リストの場合は次のような実装となる．

$$\star x = [x]$$

$$\bigoplus^{f} [x_0, x_1 \cdots x_n] = \mathrm{concat} \bigodot^{f} [x_0, x_1 \cdots x_n]$$

関数 concat はリストのリスト

$$[[x_0, x_1 \cdots x_n], [y_0, y_1 \cdots y_n] \cdots]$$

をフラットなリスト

$$[x_0, x_1 \cdots x_n, y_0, y_1 \cdots y_n \cdots]$$

へと変換する．

23.11　モナド則

新しい演算子 $\triangleleft$ を

$$(f \triangleleft g)x \xleftarrow{\mathrm{def}} \bigoplus^{f} (gx)$$

として導入する．モナドは次の規則に従う．

$$f \triangleleft \star = f \tag{23.3}$$
$$\star \triangleleft f = f \tag{23.4}$$
$$(f \triangleleft g) \triangleleft h = f \triangleleft (g \triangleleft h) \tag{23.5}$$

23.12　結合演算子

「結合演算子 $(\rightrightarrows)$」を

$$x \rightrightarrows f \xleftarrow{\mathrm{def}} \bigoplus^{f} x$$

のように定義する．

—

$$(f \triangleright g)x \xleftarrow{\mathrm{def}} (fx) \rightrightarrows g$$

$$\star \triangleright f = f \tag{23.6}$$
$$f \triangleright \star = f \tag{23.7}$$
$$(f \triangleright g) \triangleright h = f \triangleright (g \triangleright h) \tag{23.8}$$

$$\star x \rightrightarrows f = fx \tag{23.9}$$
$$[\![x]\!] \rightrightarrows \star = [\![x]\!] \tag{23.10}$$
$$([\![x]\!] \rightrightarrows f) \rightrightarrows g = [\![x]\!] \rightrightarrows (\backslash x' \to (fx' \rightrightarrows g)) \tag{23.11}$$

Or,

$$\bigoplus^{f} (\star x) = fx \tag{23.12}$$
$$\bigoplus^{\star} [\![x]\!] = [\![x]\!] \tag{23.13}$$
$$\bigoplus^{g} \left(\bigoplus^{f} [\![x]\!] \right) = \bigoplus^{x' \to (\bigoplus^{g} fx')} [\![x]\!] \tag{23.14}$$

Memo.

$$\dagger\,[\![[\![x]\!]]\!] \xleftarrow{\text{def}} [\![x]\!]$$

$$\overset{f}{\bigodot}\,[\![x]\!] = [\![x]\!] \rightrightarrows (\star \bullet f)$$

$$\dagger\,[\![x]\!] = [\![x]\!] \rightrightarrows \mathcal{O}$$

$$[\![x]\!] \rightrightarrows f \xleftarrow{\text{def}} \dagger \overset{f}{\bigodot}\,[\![x]\!]$$

$$\star \bullet f \xleftarrow{\text{def}} \overset{f}{\bigodot} \bullet \star$$

$$\dagger \bullet \overset{\dagger}{\bigodot} \xleftarrow{\text{def}} \dagger \bullet \dagger$$

$$\dagger \bullet \overset{\star}{\bigodot} \xleftarrow{\text{def}} \dagger \bullet \star = \mathcal{O}$$

$$\dagger \bullet \overset{\odot^f}{\bigodot} \xleftarrow{\text{def}} \overset{f}{\bigodot} \bullet \dagger$$

第 24 章

プログラミング*

let と where.
乱数.