第 1 章

思考の道具

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．

—

「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 2 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．

2.1 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．

2.2 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．

2.3 演算子

演算子には数学記号を割り当てる．単項演算子には論理否定 ($\neg$) とマイナス (−) がある．本書で扱う単項演算子はこの 2 種類だけである．

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 (<) である．二項演算子は多数あるので，その都度説明する．

二項演算子はたとえ積記号であっても省略できない．

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．

2.4 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．

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，括弧は冗長なのでいま後は 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

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

である．ここに記号 $\xleftarrow{\text{def}}$ は左辺を右辺で定義するという意味である．引数に「飢えた」関数 (fx) を部分適用された関数と呼ぶ．

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

$$(f \bullet g)x \xleftarrow{\text{def}} f(gx)$$

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 $f \bullet gx$ 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．

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．

$$f\,\$\,gx \xleftarrow{\text{def}} f(gx)$$

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．

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．

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

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．

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

2.5　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に予約しておきたいのと，ピリオド記号 (.) と関数合成演算子 (•) が紛らわしいため，上述の記法を用いる．

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．

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

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

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

のように書く．

記号 ◊ を用いてラムダ式をより簡潔に書くこともできる．ラムダ式 $\backslash xy \to x+y$ は

$$\Diamond + \Diamond$$

と書いても良い．式を左から読んで 1 番目の ◊ が元々の x を，2 番目の ◊ が元々の y を意味する．

2.6　関数定義

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．

関数にスペシャルバージョンがある場合は列挙する．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' は次のように定義することになる．

$$\begin{aligned} f'0 &= 0 \\ f'x &= 1+x \end{aligned}$$

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

$$\begin{aligned} f''x\,|_{x<0} &= -1 \\ |_{x\equiv 0} &= 0 \\ |_{\mathbf{otherwise}} &= 1+x \end{aligned}$$

という風に定義することにする．

2.7　条件式

条件式とはスペシャルバージョンやガードを一般化したもので，if 節と otherwise 節からなるものである．例を挙げる．

$$fx = \begin{cases} 0 & \text{if } x \equiv 0 \\ 1+x & \text{otherwise} \end{cases}$$

紙幅を節約したいときには次のように書くものとする．

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

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，

$$fx = \mathbf{if}\, px\, \mathbf{then}\, 0\, \mathbf{else}\, 1 + x$$

と書く方法もしばしば用いられる．

2.8 型

型とは変数が取りうる値に与えた制約のことである．数学者はしばしば変数 x が整数であることを $x \in \boldsymbol{Z}$ のように書くが，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

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

と書く事にする．

いま我々が考慮しておくべき型は，論理型 ($\boldsymbol{B}$)，整数型 ($\boldsymbol{Z}$)，及び実数型 ($\boldsymbol{R}$) である．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．

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

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

という型を持つ．

2.9 リスト

同じ型の値を一列に並べたものはリストである．例えば 0 から始まり 10 まで続く整数のリストは $[0,1,2,3,4,5,6,7,8,9,10]$ と書く．等差数列に限って，簡略化した書き方が許されるものとする．0 から 10 までのリストは $[0,1\cdots10]$ と書いても良い．

より高度なリスト，例えば 0 から 10 までの平方数のリストは

$$\left[x^2 \,|_{x\in[0,1\cdots10]}\right]$$

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．例えば

$$\left[x+y \,|_{x\in[0,1\cdots10]\wedge y\in[0,1\cdots5]\wedge x+y>3}\right]$$

は $0 \le x \le 10$ かつ $0 \le y \le 5$ 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

$$l +\!+ m$$

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

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

のように定義して良い．

空リストは $\emptyset$ または $[\,]$ で表す．

型 $\boldsymbol{a}$ のリストの型は $[\boldsymbol{a}]$ で表す．ある型を包み込んだ別の型を一般にコンテナ型と呼ぶ．

2.10 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．

2.11 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．

これらのペアはタプルで実装したいところだが，参照，逆参照の煩雑さを避けるために，リストのための特別な表現

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

を用い，第 2 要素である Rest はリスト型であるとする．演算子 : は結合演算子と呼ぶ．

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．

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

結合演算子 : は右結合する．すなわち

$$x_0 : x_1 : x_2 \equiv x_0 : (x_1 : x_2)$$

である．

第 3 章

型・型クラス・種

3.1 関数の型

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

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

$$\boldsymbol{Z} \to \boldsymbol{Z} \to \boldsymbol{Z}$$

である．型の式は右結合するものとする．従って上式は

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

と読む．これは関数 f が最初の引数を 1 個受け取り，$(\boldsymbol{Z} \to \boldsymbol{Z})$ 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．

3.2 型クラス

$$\boldsymbol{Z} \subset \mathbb{Q}$$

$$\boldsymbol{Z} \subset \mathbb{O}$$

$$\boldsymbol{Z} \subset \mathbb{E}$$

3.3 種

$$\boldsymbol{Z} :: *$$

$$\{\boldsymbol{Z}\} :: *$$

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

第 4 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).

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

4.1 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 $[x_0, x_1 \cdots x_n]$ の総和を

$$\sum [x_0, x_1 \cdots x_n] \xleftarrow{\text{def}} x_0 + x_1 + \cdots + x_n$$

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 $[x_0, x_1 \cdots x_n]$ が与えられたとき，

$$\bigcup_a^{\circ} [x_0, x_1 \cdots x_n] \xleftarrow{\text{def}} a \circ x_0 \circ x_1 \circ \cdots \circ x_n$$

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．

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 $\boldsymbol{a}$ 型と $\boldsymbol{b}$ 型の引数を取り $\boldsymbol{a}$ 型の戻り値を返す二項演算子，第 2（下）引数に $\boldsymbol{a}$ 型，第 3（右）引数に $\boldsymbol{b}$ 型のリストすなわち $[\boldsymbol{b}]$ 型を取り，$\boldsymbol{a}$ 型の値を返す．従って畳み込み演算子の型は

$$\bigcup :: (\boldsymbol{a} \to \boldsymbol{b} \to \boldsymbol{a}) \to \boldsymbol{a} \to [\boldsymbol{b}] \to \boldsymbol{a}$$

と定義できる．

4.2 右畳み込み

$$\bigsqcup_a^{\circ} [x_0, x_1 \cdots x_n] \xleftarrow{\text{def}} (x_0 \circ (x_1 \circ \cdots \circ (x_n \circ a)))$$

4.3 フィルタ

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

4.4 マップ

畳み込みではなく，リストはリストのままにして，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 $[x_0, x_1 \cdots x_n]$ を取り，戻り値として $[fx_0, fx_1 \cdots fx_n]$ を返す演算子 $\bigodot_{\text{list}}$ を考えよう．このとき

$$\bigodot_{\text{list}}^f [x_0, x_1 \cdots x_n] \xleftarrow{\text{def}} [fx_0, fx_1 \cdots fx_n]$$

であると定義する．リストのマップ演算子と呼ぶ $\bigodot_{\text{list}}$ は

$$\bigodot_{\text{list}} :: (\boldsymbol{a} \to \boldsymbol{b}) \to [\boldsymbol{a}] \to [\boldsymbol{b}]$$

という型を持つ．

4.5 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 $\boldsymbol{a}$ をコンテナに入れた型を一般に $\langle\!\langle \boldsymbol{a} \rangle\!\rangle$ と書くことにすると，一般化されたマップ演算子 $\bigodot$ は

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

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 $\boldsymbol{a} \to \boldsymbol{b}$ をとり $\langle\!\langle \boldsymbol{a} \rangle\!\rangle \to \langle\!\langle \boldsymbol{b} \rangle\!\rangle$ 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．

4.6 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 Nothing 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 $\{\boldsymbol{Z}\}$ と表して「きっと整数」型と呼ぶことにしよう．英語では “maybe integer” と呼ぶ．

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，一般化マップ演算子が必要である．具体的には

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

のように関数 f を「持ち上げて」から変数 $\{x\}$ に適用させる．その結果はまた $\{\boldsymbol{Z}\}$ 型である．従って，結果を受け取る変数も $\{\boldsymbol{Z}\}$ 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

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

のように書くことになるであろう．

$$\{x\}\,\{+\}\,\{y\}\,|_{\{x\}\neq\mathsf{Nothing}\wedge\{y\}\neq\mathsf{Nothing}} = \{x+y\}$$
$$|_{\mathbf{otherwise}} = \mathsf{Nothing}$$

$$\{x\}\,\{+\}\,\{y\} = \overset{\backslash x\to\bigoplus^{\backslash y\to\star(x+y)}\{y\}}{\bigoplus}\{x\}$$

4.7 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 $\boldsymbol{a}$ からリスト型 $[\boldsymbol{a}]$ を作る射 **List** ただし

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

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

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

も関手である．

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

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

を考える．型 $\langle\boldsymbol{a}\rangle$ は

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

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

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

であり，関数 g が

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

であるとき，その合成 $f \bullet g$ は

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

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

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

であるから，関数 g 並びに合成関数 $f \bullet g$ は

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

という型を持つことがわかる．今一度マップ演算子の型

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

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

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

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

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

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

である．

4.8 関手が従う規則

$$\overset{\mathsf{id}}{\bigodot} = \mathsf{id} \tag{4.1}$$

$$\overset{\mathbf{f} \bullet \mathbf{g}}{\bigodot} \equiv \overset{\mathbf{f}}{\bigodot} \bullet \overset{\mathbf{g}}{\bigodot} \tag{4.2}$$

4.9 アプリカティブ関手

一般化マップとは

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

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．

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

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

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

という型を持つ．関手の型 $\boldsymbol{a} \mapsto \langle\!\langle \boldsymbol{a} \rangle\!\rangle$ 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 $\langle\!\langle \boldsymbol{a} \rangle\!\rangle$ 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

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

4.10 モナド

ある型 $\boldsymbol{a}$ について，特殊なコンテナ $[\![\boldsymbol{a}]\!]$ 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

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

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 $\boldsymbol{a}$ をとりコンテナに収めた型 $[\![\boldsymbol{b}]\!]$ 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 $[\![\boldsymbol{a}]\!]$ をとり，戻り値としてコンテナに収められた型 $[\![\boldsymbol{b}]\!]$ を返す．

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

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

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

$[\![\boldsymbol{a}]\!]$ 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

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

であるから，関数適用された値 fx は

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

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．

4.11 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．

$$f \triangleleft \star = f \tag{4.3}$$
$$\star \triangleleft f = f \tag{4.4}$$
$$(f \triangleleft g) \triangleleft h = f \triangleleft (g \triangleleft h) \tag{4.5}$$

4.12 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．

—

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

$$\star \triangleright f = f \tag{4.6}$$
$$f \triangleright \star = f \tag{4.7}$$
$$(f \triangleright g) \triangleright h = f \triangleright (g \triangleright h) \tag{4.8}$$

$$\star x \rightrightarrows f = fx \tag{4.9}$$
$$[\![x]\!] \rightrightarrows \star = [\![x]\!] \tag{4.10}$$
$$([\![x]\!] \rightrightarrows f) \rightrightarrows g = [\![x]\!] \rightrightarrows (\backslash x' \to (fx' \rightrightarrows g)) \tag{4.11}$$

Or,

$$\bigoplus^{f} (\star x) = fx \tag{4.12}$$
$$\bigoplus^{\star} [\![x]\!] = [\![x]\!] \tag{4.13}$$
$$\bigoplus^{g} \left(\bigoplus^{f} [\![x]\!] \right) = \bigoplus^{x' \to (\bigoplus^{g} fx')} [\![x]\!] \tag{4.14}$$

Memo.

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

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

$$\dagger [\![x]\!] = [\![x]\!] \rightrightarrows \mathsf{id}$$

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

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

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

$$\dagger \bullet \overset{\star}{\odot} \xleftarrow{\text{def}} \dagger \bullet \star = \mathsf{id}$$

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

第 5 章

プログラミング

let と where.
乱数.