

関数型プログラミングと代数構造

金谷一朗

2023 年 2 月 22 日

目次

第 I 部	関数型プログラミング*	13
第 1 章	はじめに*	15
1.1	Haskell という森	15
1.2	関数型プログラミング	17
1.3	Haskell コンパイラの準備*	19
1.4	余談：本書の構成*	19
1.5	この章のまとめ*	20
第 2 章	カーリー風な書き方	21
2.1	関数	21
2.2	ラムダ式	23
2.3	パタンマッチ・ガード・条件分岐	24
2.4	余談：局所変数	27
2.5	この章のまとめ	28
第 3 章	さらにカーリー風な書き方	31
3.1	演算子	31
3.2	関数合成と関数適用	33
3.3	高階関数	34

3.4	余談：演算子の定義	35
3.5	この章のまとめ	36
第 4 章	型*	37
4.1	データ型	37
4.2	カリー化	40
4.3	多相型と型クラス*	42
4.4	余談：モノイド	45
4.5	この章のまとめ*	47
第 5 章	リスト	49
5.1	リスト	49
5.2	畳み込み	53
5.3	マップ	56
5.4	余談：リストの実装	58
5.5	この章のまとめ	59
第 6 章	再帰	61
6.1	関数の再帰適用	61
6.2	末尾再帰	62
6.3	遅延評価	64
6.4	余談：クロージャ	65
6.5	この章のまとめ	66
第 7 章	Maybe*	67
7.1	Possibly	67
7.2	Maybe	69
7.3	リストと Maybe*	73
7.4	余談: Either	75
7.5	この章のまとめ	76

第 8 章	関手*	79
8.1	圏と関手	79
8.2	アプリカティブ関手	82
8.3	関手としての関数	85
8.4	余談：アプリカティブマップ演算子の実装	88
8.5	この章のまとめ*	88
第 9 章	モナド*	89
9.1	バインド演算子	89
9.2	モナド	92
9.3	関手則・アプリカティブ関手則・モナド則	94
9.4	余談：モナドとしての関数	97
9.5	この章のまとめ*	100
第 10 章	IO*	101
10.1	アクション	101
10.2	IO モナド	103
10.3	do 記法	104
10.4	余談：関数であるということ*	106
10.5	この章のまとめ*	108
第 11 章	データ型の定義*	109
11.1	データ型	109
11.2	レコード構文	110
11.3	型クラスとインスタンス化	112
11.4	余談：レンズ*	115
11.5	この章のまとめ*	116
第 12 章	多相型の定義*	117
12.1	多相型	117

12.2	自己参照型	118
12.3	関手の拡張	119
12.4	余談: <code>newtype</code> *	120
12.5	この章のまとめ*	123
第 13 章	カテゴリ*	125
13.1	多値*	125
13.2	カテゴリ*	127
13.3	カテゴリ則*	133
13.4	余談: Kleisli 型*	133
第 14 章	アロー*	135
14.1	アロー*	135
第 15 章	プログラム	147
15.1	文字セットとコメント	147
15.2	main 関数と一般の関数定義	148
15.3	プログラミングの本質	150
15.4	余談: インタープリタ*	152
15.5	この章のまとめ*	152
第 II 部	執筆用メモ*	153
第 16 章	演算*	155
16.1	名前と予約語*	155
16.2	2 次方程式の解*	157
16.3	複素数*	159
16.4	余談: 正格評価*	162
第 17 章	より複雑な演算	163

17.1	数学関数の演算*	163
17.2	行列演算*	166
17.3	余談：アンボックス化	167
第 18 章	IO*	169
18.1	書き出し*	169
18.2	読み込み*	169
18.3	IO モナド*	169
18.4	余談：コマンドライン引数*	169
18.5	この章のまとめ*	170
第 19 章	モジュール*	171
第 20 章	代数的構造	173
20.1	数	173
20.2	群	176
20.3	圏	182
20.4	余談：束*	184
20.5	この章のまとめ	184
第 21 章	モナドとプログラミング言語*	187
21.1	ジョイン*	187
21.2	クライスリ・トリプル*	187
21.3	*	187
21.4	余談：計算可能性*	187
21.5	この章のまとめ*	187
第 22 章	ラムダ*	189
22.1	条件式	189
22.2	整数*	190

22.3	Y コンビネータ*	190
22.4	余談：冗談言語*	190
22.5	この章のまとめ*	190
第 23 章	型付きラムダ*	191
23.1	型付きラムダ*	191
23.2	カリー＝ハワード同型対応*	191
23.3	*	191
23.4	*	191
23.5	この章のまとめ*	192
第 24 章	マクロ*	193
24.1	Common Lisp のマクロ	193
24.2	Scheme のハイジェニックマクロ	193
24.3	C++ のテンプレート	193
24.4	余談：Template Haskell*	195
24.5	この章のまとめ*	198
参考文献		203

表 1 凡例 (1)

種類	字体・表記法	例
定数 有名な定数	イタリック大文字 (1 文字) ローマン大文字	<i>A, B, C</i> <i>True, False</i>
単位元	ローマンに斜線	\emptyset
変数 リスト変数 Maybe 変数 Ether 変数 一般のコンテナ変数 有名な変数	イタリック (1 文字) s をつける ? をつける ! をつける * をつける ローマン	<i>u, v, w, x, y, z</i> <i>x_s</i> <i>x?</i> <i>x!</i> <i>x*</i> <i>first, rest</i>
関数 一般のコンテナ関数 有名な関数	イタリック (1 文字) * をつける ローマン	<i>f, g, h, i, j, k</i> <i>f*</i> <i>id, sin, cos</i>
関手	フラクチュール	<i>id</i>
アクション 有名なアクション	ギリシア文字 (1 文字) スラント	<i>α, ω</i> <i>main, putStr</i>
型 有名な型 型クラス 多相型 有名な多相型	ボールド (1 文字) ボールド大文字 ボールドイタリック大文字 左肩にコンテナ名をつける 特別な括弧で包む	a, b, c Bool, Int <i>Eq, Ord</i> Maybe [a] [a], [[a]]
型コンストラクタ 有名な型コンストラクタ	ボールドイタリック ローマン大文字	<i>f, g, h</i> List, Maybe
集合	ブラックボード大文字	$\mathbb{B}, \mathbb{Z}, \mathbb{R}$

表 2 凡例 (2)

種類	字体・表記法	例
値コンストラクタ 有名な値コンストラクタ	名前付き括弧で包む 特別な括弧で包む	$\text{Right} \llbracket x \rrbracket, \text{Left} \llbracket x \rrbracket$ $[x], \llbracket x \rrbracket, \langle x \rangle, \langle\langle x \rangle\rangle$
キーワード do 記法中のキーワード	サンセリフ タイプライタ	if, otherwise let
リスト 集合 タプル	ブラケットで包む ブレースで包む 丸括弧で包む	$[x, y]$ $\{x, y\}$ $\begin{pmatrix} x \\ y \end{pmatrix}$

表 3 記号一覧 (1)

記号	意味	Haskell 記号
\backslash, \mapsto \diamond	ラムダ式 無名パラメタ	\backslash, \rightarrow
$::$ \in \rightarrow	型集合の元 値集合の元 写像	$::$ $<-$ \rightarrow
★ +	任意の二項演算子 一般加法演算子	
$\dot{+}$	複素数値コンストラクタ	$:+$
$\triangle_{\text{Integral}}$ ★	型変換演算子 種	<code>fromIntegral</code> ★
▪ ● \$	関数合成 圏の関数合成 関数適用	\cdot \cdot \$
\bigcup \bigsqcup	左畳み込み 右畳み込み	<code>foldl</code> <code>foldr</code>
$:$ \oplus \otimes	<code>cons</code> 演算子 結合演算子 ジップ演算子	$:$ <code>++</code> <code>zip</code>
\flat # \flat	リスト平坦化演算子 Maybe 平坦化演算子 ジョイン演算子	<code>concat</code> <code>join</code> <code>join</code>
$[]$ \emptyset \emptyset	空リスト 空 Maybe (ナッシング) 空	<code>[]</code> <code>Nothing</code>

表 4 記号一覧 (2)

記号	意味	Haskell 記号
$\llbracket x \rrbracket$	ピュア演算子	<code>pure x</code>
$[x]$	リスト値コンストラクタ	<code>[x]</code>
<code>Just $\llbracket x \rrbracket$</code>	Maybe 値コンストラクタ	<code>Just x</code>
<code>Left $\llbracket x \rrbracket$, Right $\llbracket x \rrbracket$</code>	Either 値コンストラクタ	<code>Left x, Right x</code>
$\langle x \rangle$	定数演算子	<code>const x</code>
$\langle\langle x \rangle\rangle$	アロー演算子	<code>arr x</code>
\cdot	一般マップ	<code><\$></code>
\odot	リストのマップ	<code><\$></code>
\square	Maybe のマップ	<code><\$></code>
\circ	関数のマップ	<code><\$></code>
\times	一般アプリカティブマップ	<code><*></code>
\otimes	リストのアプリカティブマップ	<code><*></code>
\boxtimes	Maybe のアプリカティブマップ	<code><*></code>
\bowtie	関数のアプリカティブマップ	<code><*></code>
\heartsuit	一般バインド演算子	<code>=<<</code>
\clubsuit	リストのバインド演算子	<code>=<<</code>
\spadesuit	Maybe のバインド演算子	<code>=<<</code>
\diamond	関数のバインド演算子	<code>=<<</code>
\bowtie	バインド・合成演算子	<code><=<</code>
$\heartsuit \rightarrow$	右バインド演算子	<code>>>=</code>
\rightarrow	右バインド演算子 (戻値無視)	<code>>></code>
\llcorner	カテゴリ関数合成	<code><<<</code>

第Ⅰ部

関数型プログラミング*

第 1 章

はじめに*

本書はプログラミング言語 Haskell の入門書である．それと同時に，本書はプログラミング言語を用いた代数構造の入門書でもある．プログラミングと代数構造の間には密接な関係があるが，特に関数型プログラミングを実践する時にはその関係を意識する必要がある．本書はその両者を同時に解説することを試みる．

1.1 Haskell という森

これからのプログラマにとって Haskell を無視することはできない．Haskell の「欠点をあげつらうことも，攻撃することもできるが，無視することだけはできない」のだ．それは Haskell がプログラミングの本質に深く関わっているからである．

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。そこで、本書では近代的なプログラマを対象に、プログラミング言語を登山口に抽象数学の山を登り、その高みから Haskell という森を見下ろすことにする。

さて、登山口にどのプログラミング言語を選ぶのが適当であろうか。TIOBE Index 2023 年 2 月版によると「ビッグ 5」として Python, C, C++, Java, C# が挙げられている。^{*1}順位の変動はあるが、他の調査でもビッグ 5 は過去何年も変動していないので、当座は妥当な統計であろう。このうち C は「多くのプログラマが読める」以外にメリットが無く、その唯一のメリットさえ最近は怪しくなっているため、登山口候補から外す。残るは Java, C++, C# グループと Python ということになるが、シンプルであり、かつ Haskell と対極にある言語である Python を登山口に選ぶことにした。

本書では Python コードはこのように登場する。

Python

```
print("Hello, world.")
```

本書に示すコードは擬似コードではなく、すべて実行可能な本物のコードである。

ところで、一部の章でどうしても型に触れないといけない部分がある。Python は動的型付け言語であり、型の説明には不適切であるため、この部分だけ理解の助けとして C++17 によるコードを例示した。この部分はコードを読まなくても先に進める。

* * *

ところで、プログラムのソースコードは現代でも ASCII 文字セットの範囲で書くことが標準的である。Unicode を利用したり、まして文字にカラー

^{*1} <https://www.tiobe.com/tiobe-index/>

を指定したり、書体や装飾を指定することは一般的ではない。例えば変数 a のことを \mathbf{a} と書いたり \underline{a} と書いたり \hat{a} と書いたりして区別することはない。

Haskell プログラマもまた、多くの異なる概念を同じ貧弱な文字セットで表現しなければならない。これは、初めて Haskell コードを読むときに大きな問題になり得る。例えば Haskell では $[a]$ という表記をよく扱う。この $[a]$ は a という変数 1 要素からなるリストのこともあるし、 a 型という仮の型から作ったリスト型の場合もあるが、字面からでは判断できない。もし変数はイタリック体、型はボールド体と決まっていれば、それぞれ $[a]$ および $[a]$ と区別出来たところである。

本書は、異なる性質のものには異なる書体を割り当てるようにしている。ただし、どの表現もいつでも Haskell に翻訳できるように配慮している。実際、本書執筆の最大の困難点は、数学的に妥当で、かつ Haskell の記法とも矛盾しない記法を見つけることであった。

1.2 関数型プログラミング

プログラマはなぜ Haskell を習得しなければならないのだろう。それは Haskell と 関数型プログラミング の間に密接な関係があるからである。

関数型プログラミングとはプログラミングにおける一種のスローガンのようなもので、どの言語を用いたから関数型でどの言語を用いたから関数型ではない、というものではない。しかし、関数型プログラミングを強くサポートする言語と、そうでない言語とがある。こちら辺の事情はオブジェクト指向プログラミングとプログラミング言語の関係と似ている。Haskell は関数型プログラミングを強くサポートし、Python はほとんどサポートしない。

関数型プログラミングの特徴を一言で言えば、プログラム中の 破壊的代入 を禁止することである。変数 x に 1 という数値が一度代入されたら、変数 x の値をプログラム中に書き換える、すなわち破壊的代入をすることはできない。この結果、変数の値はプログラムのどこでも、どの時点で読み出しても同じであることが保証される。これを変数の 参照透過性 と呼ぶ。

プログラム全体に参照透過性があると、そのプログラムはブロックに分割しやすく、各々のブロックは再利用しやすい。またプログラムのどの断片から読み始めても、全体の構造を見失いにくい。これが関数型プログラミングとそれを強くサポートする Haskell を習得する理由である。

参照透過性がもたらすもう一つのボーナスは変数の遅延評価である。変数はいつ評価しても値が変わらないのだから、コンパイラは変数をできるだけ遅く評価してよい。この遅延評価によって、Haskell コンパイラは他の言語に見られない無限リストを扱う能力を獲得している。

* * *

ここで数学とプログラミングの関係について述べておこう。ある方程式を解くためにコンピュータによって数値シミュレーションを行うとか、非常に複雑な微分を機械的に行うとか、プログラミングによって数学をサポートすることは計算機科学の主たる分野の一つであるが、ここではもっと根源的な話をする。

数学者もプログラマも関数をよく使う。数学者が使う関数とは、引数がいくつかあって、その結果決まる戻り値があるようなものだ。一方でプログラマが使う関数というのは、引数と戻り値はだいたい同じとして、中身に条件分岐があったり、ループがあったり、外部変数を書き換えたり、入出力をしたりする。

どちらも同じ関数であるのに、なぜこうもイメージが違うのだろうか。

もし、我々が関数型プログラミングの原則を忠実に守り、プログラム中のいかなる破壊的代入をも禁止するとすると、両者の関数は全く同じ性格になる。逐次実行も条件分岐もループも、それどころか定数さえ、ラムダ式という式だけで書けるようになる。あらゆるプログラムが、最終的には単一のラムダ式で書ける。

ところが、入出力、状態変数、例外など、プログラミングに使われる多くのテクニックは関数の副作用を前提としている。参照透過性と副作用を統一的に扱うためにはモナドという数学概念が必要である。Haskell はモナドを

陽に扱うプログラミング言語である。

1.3 Haskell コンパイラの準備*

本書の第 I 部は Haskell コンパイラ無しで読み進めることができる。とは言え、前もって Haskell コンパイラを用意しておくことは無意味とも言えない。

~~TK: To be written.~~

1.4 余談：本書の構成*

ところで、各章の終りにはこのような「余談」の節を設けている。余談には、本書を読み進めるに当たって本質的ではない話を詰め込んでいる。ではなぜ書くかというと、これは筆者が頭を整理するために書くのである。なので、筆者が何を思って本書を執筆していたかを知りたいときが万が一来れば、目を通してもらいたい。

最初の余談は本書の構成である。

そもそも本の構成など、目次を見ればわかることであるが、執筆時点では目次がまだない。そこで、筆者がときどきこの節を読み返し、全体の構成を確認しているのである。

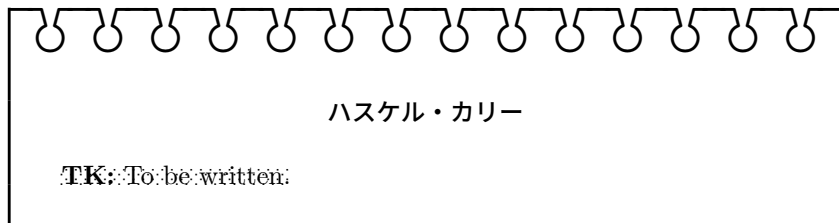
本書は 3 部構成になっている。第 I 部はプログラミング言語から抽象数学への登山である。第 II 部は抽象数学の山頂から Haskell の森へ下っていく道である。第 III 部はこれまで歩いた道全体を俯瞰する。

第 I 部は対応する Haskell のコードを脚注に記載している。脚注部分は Haskell を一通り覚えた後で読み返してもらいたい。^{*2}

^{*2} Haskell のコードは `x = 1` のようにタイプライタ体で書く。

1.5 この章のまとめ*

1. ...
2. 各章にはこのように「この章のまとめ」を置く.



第 2 章

カーリー風な書き方

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 Haskell に近い記法から始めることにする。

2.1 関数

数学やプログラミング言語には書き方に一定の決まりがある。この章ではまず「カーリー風の」数式記述方式を見てみることにする。「カーリー風」というのは、数学者ハスケル・カーリーから名前を借りた言い方で、筆者が勝手に命名したものだ。

カーリー風の書き方は数学の教科書やプログラミングの教科書で見かけるものとは若干違うが、圧倒的にシンプルで Haskell との親和性も高く、慣れてくると非常に読みやすいものなので、本書でも全面的に採用する。

まずは関数から見ていくことにしよう。Python や一般的な数学書では引数 x をとる関数 f を

$f(x)$

Python

と書くが、括弧は冗長なので今後は

$$fx \quad (2.1)$$

と書くことにする。^{*1}

関数 f に引数 x を「食わせる」ことを関数適用と呼ぶ。もし fx と書いてあったら、それは f と x の積、つまり $f * x$ ではなく、従来の $f(x)$ すなわち関数 f に引数 x を与えているものと解釈する。高校生向けの数学書でも $\sin x$ のように三角関数に限ってはカリー風を書くことになっているので、まるで馴染みがないということもないだろう。なお、関数はいつも引数の左側に書くことにする。これを「関数 f が変数 x の左から作用する」と言い、また関数 f のことを左作用素とも呼ぶ。

複数引数をとる関数を Python や一般的な数学の教科書では

$g(x, y)$

Python

と書くが、これも括弧が冗長なので今後は gxy と書く。この場合式 gxy は左を優先して結合する。つまり

$$gxy = (gx)y \quad (2.2)$$

である。これは引数 y に関数 (gx) が左から作用していると解釈する。関数 (gx) は引数 x に関数 g を作用させて作った関数である。引数に「飢えた」関数 (gx) を部分適用された関数と呼ぶ。

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。Haskell の場合、関数適用はいつも左結合である。

^{*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。

部分適用の例を見てみよう．例えばふたつの引数のうち大きい方を返す関数 `max` は

$$\text{max } xy \quad (2.3)$$

として使われるが，関数適用は左結合であるから

$$(\text{max } x)y \quad (2.4)$$

としても同じである．そこで $(\text{max } x)$ だけ取り出すと，これは「引数が x よりも小さければ x を，そうでなければ引数を返す関数」とみなすことができる．^{*2}

2.2 ラムダ式

関数の正体はラムダ式である．ラムダ式とは，仮の引数を取り，その値をもとにながしかの演算を行い，その結果を返す式である．ラムダ式は名前のない関数のようなものだ．それゆえ，無名関数と呼ばれることもある．

例えば引数 x をとり値 $1 + x$ を返すラムダ式を Python では

```
lambda x: 1+x
```

Python

と書くが，我々はより簡潔に

$$\backslash x \mapsto 1 + x \quad (2.5)$$

と書くことにする．この式は多くの書物で $\lambda x. 1 + x$ 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 (.) が今後登場する二項演算子 \cdot と紛らわしいため，上述の記法を用いる．^{*3}

^{*2} Haskell では `max xy` を `max x y` と書く．

^{*3} Haskell ではラムダ式 $\backslash x \mapsto 1 + x$ を `\x -> 1+x` と書く．ラムダ式は元々は $\hat{x}. x + 1$

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．

$$(\backslash x \mapsto 1 + x)2 \quad (2.6)$$

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

$$\backslash xy \mapsto x + y \quad (2.7)$$

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 \diamond 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 \diamond の部分には引数が入る．第 n 番目の \diamond には第 n 番目の引数が入る．例えばラムダ式 $\backslash xy \mapsto x + y$ は

$$\diamond + \diamond \quad (2.8)$$

と書いても良い．式を左から読んで 1 番目の \diamond が元々の x すなわち第 1 引数を，2 番目の \diamond 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

2.3 パターンマッチ・ガード・条件分岐

関数の定義は，基本的にはラムダ式の変数への代入である．引数 x をとり値 $(\sin x)/x$ を返す関数 f は

$$f = \backslash x \mapsto (\sin x)/x \quad (2.9)$$

のように書かれていた．これが次第に $\wedge x. x + 1$ となり， $\lambda x. x + 1$ に変化していったと言われている．Haskell が λ の代わりに \backslash 記号を使うのは，その形が似ているからである．

*4 この記法は Haskell にはない．

と定義できる。ただし、この省略形として

$$f\,x = (\sin x)/x \quad (2.10)$$

と書いても良い。この書き方は今後頻出するので、是非覚えておいてもらいたい。^{*5}

* * *

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が 1 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

$$\begin{cases} f'\,0 = 1 \\ f'\,x = (\sin x)/x \end{cases} \quad (2.11)$$

ように定義する。これを関数のパターンマッチと呼ぶ。^{*6}

関数のパターンマッチは、関数の内部に書いても良い。関数内部にパターンマッチを書きたい場合は

$$f'\,x = \text{case } x \text{ of } \begin{cases} 0 & \text{-->} 1 \\ - & \text{-->} (\sin x)/x \end{cases} \quad (2.12)$$

のように書く。ここに $-$ は任意の値の意味である。パターンマッチは上から順番にマッチングしていくため、この場合は 0 以外を意味する。^{*7}

^{*5} Haskell では $f = \backslash x \mapsto (\sin x)/x$ を $f = \backslash x \rightarrow (\sin x)/x$ と書き、一方 $f\,x = (\sin x)/x$ を $f\,x = (\sin x)/x$ と書く。

^{*6} Haskell では

$$f'\,0 = 1 \quad f'\,x = (\sin x)/x$$

と書く。Haskell ではプライム記号 (') はアルファベットと同一視される。

^{*7} Haskell では

$$f'\,x = \text{case } x \text{ of } \begin{cases} 0 & \text{-->} 1 \\ - & \text{-->} (\sin x)/x \end{cases}$$

または $f'\,x = \text{case } x \text{ of } \{0 \rightarrow 1; - \rightarrow (\sin x)/x\}$ と書く。

一部のプログラミング言語ではデフォルト引数という、引数を省略できるメカニズムがあるが、我々は引数をいつも省略しないことにする。^{*8}

* * *

関数定義にパターンマッチではなく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 0 を、 0 の場合は 1 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

$$\begin{aligned} f''x \mid x < 0 &= 0 \\ \mid x = 0 &= 1 \\ \mid \text{otherwise} &= (\sin x)/x \end{aligned} \tag{2.13}$$

という風に定義する。^{*9}

* * *

関数定義の場合分けを駆使すれば条件式はなくても構わないが、条件式の記法があるのは便利である。Python には

Python

```
def f(x):
    if x==0:
        return 1
    else:
        return sin(x)/x
```

のような制御構造としての条件文があるが、我々は値を持つ条件式を考える。

^{*8} Haskell にもデフォルト引数はない。

^{*9} Haskell では

```
f'' x | x < 0 = 0
      | x == 0 = 1
      | otherwise = (sin x)/x
```

と書く。

我々の条件式とは

$$fx = \text{if } x \equiv 0 \text{ then } 1 \text{ else } (\sin x)/x \quad (2.14)$$

のように if 節, then 節, 及び else 節からなるものであって, then 節も else 節も省略できないものとする. if 節の式の値が真 (True) であれば then 節の式が評価され, 偽 (False) であれば else 節の式が評価される. 我々の条件式は C/C++ における条件演算子 (三項演算子) と等しく見えるが, Haskell の場合は遅延評価が行われるため, 結果として条件式の短絡評価が行われる点が異なる.*¹⁰

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

$$p = (\diamond \equiv 0) \quad (2.15)$$

としておき,

$$fx = \text{if } px \text{ then } 1 \text{ else } (\sin x)/x \quad (2.16)$$

と書く方法もしばしば用いられる.

2.4 余談：局所変数

数式が長く続くとき, 読みやすさのために局所変数を導入すると便利である. 例えば

$$y = f(1 + x) \quad (2.17)$$

という式のうち, 先に $1 + x$ の部分を計算して x' のように名前をつけておきたいこともあるであろう. そんなときは

$$y = \text{let } x' \triangleq 1 + x \text{ in } fx' \quad (2.18)$$

*¹⁰ Haskell では $fx = \text{if } x \equiv 0 \text{ then } 1 \text{ else } (\sin x)/x$ を $f\ x = \text{if } x == 0 \text{ then } 1 \text{ else } (\sin x)/x$ と書く.

と書くことにする．このようにして導入された x' を 局所変数 と呼ぶ.^{*11}

式 (2.18) は局所変数を後ろに回して

$$y = f x' \text{ where } x' \triangleq 1 + x \quad (2.19)$$

のように書いても良い.^{*12}

式 (2.18) や式 (2.19) はラムダ式を使った 糖衣構文 (シンタックスシュガー) であり、一般に

$$\text{let } x' \triangleq gx \text{ in } f x' = f x' \text{ where } x' \triangleq gx \quad (2.20)$$

$$= (\backslash x' \mapsto f x')(gx) \quad (2.21)$$

である．ただし let 節が式であるのに対し，where 節は式ではないことに気をつけよう．

2.5 この章のまとめ

1. 変数 x に関数 f を適用することを $f x$ と書く．
2. 変数 x, y に関数 g を適用することを $g x y$ と書く．関数適用は左結合するので $g x y = (g x) y$ である．この関数 $(g x)$ は部分適用された関数と呼ぶ．
3. 関数はラムダ式で定義する．ラムダ式は $\backslash x \mapsto (\sin x)/x$ のように表記する．
4. ラムダ式は $f \diamond$ のように無名パラメタ \diamond を用いて表記しても良い．
5. 関数定義 $f = \backslash x \mapsto (\sin x)/x$ は $f x = (\sin x)/x$ と省略表記できる．
6. 関数定義にはパターンマッチが使える．例えば $f' 0 = 1; f' x = (\sin x)/x$ と定義できる．
7. パターンマッチは関数本体に書いても良い．関数 f' の例で言えば $f' = \text{case } x \text{ of } 0 \rightarrow 1; _ \rightarrow (\sin x)/x$ と定義できる．

^{*11} Haskell では $y = \text{let } x' \triangleq 1 + x \text{ in } f x'$ を $y = \text{let } x' = 1 + x \text{ in } f x'$ と書く．

^{*12} Haskell では $y = f x' \text{ where } x' \triangleq 1 + x$ は $y = f x' \text{ where } x' = 1 + x$ と書く．

8. 関数定義にはガードが使える．例えば $f''|_{x<0} = 0; f''|_{x\equiv 0} = 1; f''|_{\text{otherwise}} = (\sin x)/x$ と定義できる．
9. 条件分岐は `if x then y else z` と書き， $x \equiv \text{True}$ の時には y が， $x \equiv \text{False}$ のときには z が式の値になる．
10. 局所変数は $y = \text{let } x' \triangleq \dots \text{ in } f x'$ または $y = f x' \text{ where } x' \triangleq \dots$ という書き方で導入できる．

式の評価順序

Haskell は参照透過な言語なので，式がいつ評価されるかを考える必要はない．一方で，参照透過でない言語は式の評価順序をいつも気にしておく必要がある．例えば C は関数引数の評価順序を定めていないので，次のコード

```
int i = 0;
printf("%d, %d\n", ++i, ++i);
```

C

は画面に 1, 2 を出力する場合もあるし，2, 1 を出力する場合もある．

第3章

さらにカーリー風な書き方

我々は関数とラムダ式の「カーリー風」な書き方を見てきた。この章ではさらに演算子、関数合成についても「カーリー風」な書き方を見ていく。

3.1 演算子

演算子は関数の特別な姿である。演算子は作用素と呼んでも良い。どちらも英語の operator の和訳である。演算子は普通アルファベット以外のシンボル 1 個で表現し、変数や関数の前に置いて直後の変数や関数に作用させるか、2 個の変数や関数の間に置いてその両者に作用させる。例えば $-x$ のマイナス記号 ($-$) は変数の前に置いて直後の変数 (x) に作用する演算子であり、 $x + y$ のプラス記号 ($+$) は 2 個の変数の間に置いてその両者 (x, y) に作用する。

1 個の変数または関数に作用する演算子を単項演算子と呼び、2 個の変数または関数に作用する演算子を二項演算子と呼ぶ。本書では単項演算子はすべて変数の前に置く、すなわち前置する。前置する演算子のことを前置演算子と呼ぶが、数学者は同じものを左作用素と呼ぶ。

Haskell には単項マイナス ($-$) を除いて他に単項演算子はない。

二項演算子のうちよく使われるものは和 ($+$)、積 ($*$)、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.*1

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 \star について $x \star y$ 及び $(\star)xy$ は全く同じ意味である. すなわち

$$(\star)xy = x \star y \quad (3.1)$$

である. 従って, 二項演算子と2引数関数に本質的な差はない. 本書では演算子と関数という用語は全く同じ意味で用いる.*2

一般の関数が左結合であることを思い出すと, 二項演算子を関数に見立てた (\star) も

$$(\star)xy = ((\star)x)y \quad (3.2)$$

であるから, 部分適用が可能である. 式 (3.2) から第2引数 y を取り除いて $(\star)x$ という「餓えた」1引数関数を取り出せる. 例えば関数 $((+)1)$ は引数に1を加える関数である.*3

* * *

二項演算子の部分適用に限ってセクションと言う記法も用いられる. 二項演算子 \star に対して $(\star x)$ および $(x \star)$ はそれぞれ

$$(\star x) = \diamond \star x \quad (3.3)$$

$$(x \star) = x \star \diamond = (\star)x \quad (3.4)$$

*1 Haskell では \wedge を $\&\&$ と書き, \vee を $||$ 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2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.

*3 Haskell では $((+)1)$ を $((+)1)$ と書く.

である。例えば $(1+)$ は $((+)1)$ と等価であり、これは $(+1)$ と同等である。ただし、マイナス演算子 $(-)$ だけは例外で、 (-1) はマイナス 1 を表す。負の数をいつも括弧で包んでおくのは良いアイデアである。^{*4}

なお、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。また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。

3.2 関数合成と関数適用

ある変数に複数の関数を順に適用することはよくあることである。例えば

```
y = f(x)
z = g(y)
```

Python

あるいは、同じことであるが

```
z = g(f(x))
```

Python

とすることがある。本書の記法で書けば

$$z = g(fx) \tag{3.5}$$

である。式 (3.5) から括弧を省略して $z = gfx$ としてしまうと、関数適用は左結合するから $z = (gf)x$ の意味になってしまう。関数 g が引数に関数を取るのではない限り (gf) は無意味なので、式 (3.5) の括弧は省略できない。

ここで、引数のことは忘れて、関数 f と関数 g を先に合成しておきたいとしよう。その合成を $g \cdot f$ と書く。演算子 \cdot は関数合成演算子と呼ぶ。合成はラムダ式を使って

$$g \cdot f = g(f\circ) \tag{3.6}$$

^{*4} Haskell では $(1+)$ を $(1+)$ と書く。また (-1) はセクションではなくマイナス 1 を表す (-1) というリテラルとみなされる。ただし $(- 1)$ のように空白を挟んでも同じくマイナス 1 とみなされる (1 というリテラルに単項マイナス演算子が適用される)。

と定義できる。関数合成演算子 \cdot は関数適用よりも優先順位が高く、 $(g \cdot f) x$ は単に $g \cdot f x$ 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。式 (3.5) は関数合成演算子を用いると

$$z = g \cdot f x \quad (3.7)$$

と書ける。^{*5}

関数合成演算子とは逆に、結合の優先順位の低い関数適用演算子も考えておくと便利なこともある。関数適用演算子 $\$$ を次のように定義しておく。

$$f \$ x = f x \quad (3.8)$$

演算子 $\$$ の優先順位は関数適用も含めあらゆる演算子よりも低いものとする。関数適用演算子を用いて式 (3.5) を書き直すと

$$z = g \$ f x \quad (3.9)$$

となる。演算子 $\$$ の優先順位は足し算よりも低いので $f(x+1)$ は $f \$ x + 1$ と書くこともできる。演算子 $\$$ を閉じ括弧のいらない開き括弧と考えてもよい。^{*6}

関数適用演算子のもう一つの興味深い使い方は、関数適用演算子の部分適用である。セクション $(\$ x)$ を用いると

$$(\$ x) f = f \$ x \quad (3.10)$$

であるから、関数適用演算子を用いて引数を関数に渡すことができる。^{*7}

3.3 高階関数

関数を引数に取ったり、あるいは関数を返す関数のことを高階関数と呼ぶことがある。関数合成演算子と関数適用演算子は高階関数の好例である。

^{*5} Haskell では関数 f_2 と関数 f_1 の合成は $f_2 \cdot f_1$ である。式 $z = g \cdot f x$ は $z = f_2 \cdot f_1 x$ と書く。

^{*6} Haskell では $g \$ f x$ を $f_2 \$ f_1 x$ と書く。

^{*7} Haskell では $(\$ x) f$ を $(\$ x) f$ と書く。

他に例えば、引数として整数 a を取り、関数 $fx = a + x$ を返すような関数 g を

$$ga = a + \diamond \quad (3.11)$$

のように定義することも可能である。このとき、

$$f = g\ 100 \quad (3.12)$$

$$x = f\ 1 \quad (3.13)$$

とすれば $x = 101$ を得る。^{*8}

高階関数は今後度々顔をだすことになる。第 5 章に登場するマップ演算子や畳込み演算子は高階関数の一種である。

3.4 余談：演算子の定義

Haskell では関数だけでなく、新しい演算子も定義できる。^{*9}

計算機科学者 Donald Knuth は、整数 x, y が与えられたとき x の y 乗を x^y ではなく $x \uparrow y$ と書いた。これは

$$x \uparrow y = \underbrace{x * x * \cdots * x}_y \quad (3.14)$$

という意味である。^{*10}

Knuth はさらに演算子 $\uparrow\uparrow$ を

$$x \uparrow\uparrow y = \underbrace{x \uparrow x \uparrow \cdots \uparrow x}_y \quad (3.15)$$

^{*8} Haskell では $ga = a + \diamond$ を $ga = \backslash x \mapsto a + x$ と展開しておいて $g\ a = \backslash x \rightarrow a + x$ と書く。

^{*9} Haskell で演算子に使える記号は

! @ # \$ % ^ & * - + = . \ | / < : > ? ~

の組み合わせである。

^{*10} Haskell では $x \uparrow y$ を x^y と書く。

のように定義した．これをクヌースの矢印と呼ぶ．クヌースの矢印は

$$x \uparrow\uparrow n \mid_{n \leq 0} = 1 \quad (3.16)$$

$$\mid_{\text{otherwise}} = x \uparrow (x \uparrow\uparrow (n - 1)) \quad (3.17)$$

と定義できる．*11

なおこの定義は自分自身を呼び出す再帰を行っている．再帰に関しては第6章で詳しく述べる．

3.5 この章のまとめ

1. 関数と演算子は同じものである．
2. 任意の二項演算子 \star について $x \star y$ と $(\star)xy$ は全く同じ意味である．
3. 任意の二項演算子 \star について $(\star x) = \diamond \star x$ であり, $(x \star) = x \star \diamond$ である．これらの記法をセクション記法と呼ぶ．
4. 関数 f と関数 g を合成した関数を $f.g$ と書く．
5. 式 $f \$ gx$ は $f(gx)$ の意味である．
6. 関数を引数に取ったり，関数を返す関数のことを高階関数と呼ぶ．
7. Haskell のラムダ式はいつもレキシカルクロージャである．

*11 Haskell では

```
x^^.n | n <= 0 = 1 | otherwise = x^(x^^.(n-1))
```

と書く．演算子 ^^ が予約済みのため ^^. を使った．なお厳密には

```
(^^.) :: Integral a => a -> a -> a
```

と演算子の型を宣言しておく必要がある．型に関しては第4章で述べる．

第 4 章

型*

Haskell の変数、関数にはすべて型がある。プログラマの言う型とは、数学者の言う集合のことである。本章では、Haskell が扱う基本的な型であるデータ型と、パラメトリックな型である多相型、および型の型である型クラスについて述べる。また関数のカリー化についても述べる。

4.1 データ型

型とは変数を取りうる値に言語処理系が与えた制約のことである。Haskell を含む多くのコンパイラ言語は静的型付けと言って、コンパイル時までに変数の型が決まっていることをプログラマに要求する。一方、Python のようなインタプリタ言語はたいてい動的型付けと言って、プログラムの実行時まで変数の型を決めない。

変数に型の制約を設ける理由は、プログラム上のエラーが減ることを期待するためである。例えば真理値が必要とされるところに整数値の変数が来ることは悪い予兆である。一方で C 言語のように全ての変数にいちいち型を明記していくのも骨が折れる。

数学者や物理学者は変数に型の制約を求める一方、新しい変数の型は明記せず読者に推論させる方法をしばしばとる。例えば、質量 m は「スカラー」

という型を持つし、速度 v は「3次元ベクトル」という型を持つ。スカラーと3次元ベクトルの間に足し算は定義されていないため、例えば $m + v$ という表記を見たときに、両者の型を知っていれば直ちにエラーであることがわかる。

Haskell はコンパイラが型推論を行うことで、型が自明の場合は型を省略することが出来る。

* * *

Haskell にはよく使う型が予め用意されている。例えば論理型 は論理値すなわち真 (**True**) または偽 (**False**) という値をとる変数の型である。ある変数 x が論理型であることを、Haskell では

$$x :: \text{Bool} \tag{4.1}$$

と書く。数学者なら同じことを

$$x \in \mathbb{B} \tag{4.2}$$

と書くところであるが、ここは Haskell の流儀に従おう。また型定義と値定義はよく一緒に行われるので、今後

$$\left\| \begin{array}{l} x :: \text{Bool} \\ x = \text{True} \end{array} \right. \tag{4.3}$$

のようにまとめて書くことにしよう。^{*1}

この場合変数 x が **Bool** 型であることは自明であるため $x :: \text{Bool}$ は省略できるが、可能な限り型を明記しておくことは良い習慣である。

^{*1} Haskell では

```
x :: Bool
x = True
```

と書く。

他に整数を表す整数型がある。整数型には 2 種類あって、その一つは **Int** である。この **Int** は C の `int` と似た「計算機にとって都合の良い整数」である。計算機にとって都合の良い整数とは、例えば 64 ビット計算機の場合 -2^{63} から $2^{63} - 1$ の間の整数という意味である。^{*2}

整数型には **Integer** もある。この **Integer** は計算機にとっては非常識なぐらい大きな、あるいは小さな値を表すことができる。^{*3}

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（単精度）の浮動小数点数型である **Float** と、倍精度浮動小数点数型である **Double** が提供される。^{*4}

もう一つ、計算機ならではの型がある。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文字型 **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。^{*5}

理由の第一は、文字が小さな整数であるため、文字型を独立して定義しておくことでメモリを節約できるのである。特にメモリが高価であった時代はこれが唯一の理由であった。現在でも、整数が一般に 64 ビットを消費するのに対し、UTF8 文字エンコードを用いている場合、アルファベットは 8 ビットしか消費しない。

理由の第二は、単純に整数と文字が異なるからである。文字を表す変数に整数を代入するのは悪い兆しである。

理由の第三は、文字が数値にエンコードされる方式が可変長である場合に備えて、整数と区別しておくためである。例えば UTF8 文字エンコードは可変長エンコーディングを行う。

このような基本的な型をデータ型と呼ぶ。

^{*2} Haskell では **Int** を `Int` で表す。

^{*3} Haskell では **Integer** を `Integer` で表す。

^{*4} Haskell では **Float**, **Double** をそれぞれ `Float`, `Double` で表す。

^{*5} Haskell では **Char** を `Char` で表す。

* * *

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

$$f :: \text{Int} \rightarrow \text{Int} \quad (4.4)$$

という型を持つ。上式は

$$f :: \underbrace{\text{Int}}_x \rightarrow \underbrace{\text{Int}}_{fx} \quad (4.5)$$

のようにイメージすると良い。これは関数 f が集合 Int から集合 Int への写像であると読む。^{*6}

4.2 カリー化

Haskell では、どのような関数であれ引数は 1 個しかとらない。引数が 2 個あるように見える関数として、例えば gxy があったとしよう。ここに g は関数、 x, y は変数である。関数適用は左結合であるから、これは $(gx)y$ である。ここに (gx) は引数 y をとる関数であると見ることができる。つまり、関数 f とは引数 x をとり「引数 y をとって値を返す関数 (gx) を返す」関数であると言える。

二項演算 $x + y$ は $(+)xy$ とも書けたことを思い出そう。これも左結合を思い出すと

$$(+)xy = ((+)x)y \quad (4.6)$$

であるから、 y という引数を $((+)x)$ という関数に食わせていると解釈できる。

ラムダ式の場合は話はもっと単純で、形式的に

$$\backslash xy \mapsto x + y = \backslash x \mapsto (\backslash y \mapsto x + y) \quad (4.7)$$

^{*6} Haskell では $f :: \text{Int} \rightarrow \text{Int}$ と書く。

のように展開すれば 1 引数にできる. 矢印 \mapsto は右結合である. そこでこのラムダ式は括弧を省略して

$$\backslash xy \mapsto x + y = \backslash x \mapsto \backslash y \mapsto x + y \quad (4.8)$$

とも書かれる.

複数引数をとる関数を 1 引数関数に分解することをカリー化と呼ぶ. これはこの分野の先駆者であるハスケル・カリーの名前に由来する.

整数引数を二つ取り, 整数を返す関数 g は

$$g :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad (4.9)$$

という型を持つ. 写像の矢印記号は右結合するので, これは

$$g :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \quad (4.10)$$

と同じ意味である. 上式は

$$g :: \underbrace{\text{Int}}_x \rightarrow \underbrace{\text{Int} \rightarrow \text{Int}}_{gx} \quad \begin{array}{c} y \\ \underbrace{\phantom{\text{Int} \rightarrow \text{Int}}} \end{array} \quad \begin{array}{c} (gx)y \\ \underbrace{\phantom{\text{Int}}} \end{array}$$

のようにイメージすると良い. 自然言語で考えると **Int** 型の引数を一つ取り, **Int** 型の引数を一つ取って **Int** 型の値を返す関数を返す, と読める.*⁷

* * *

Haskell にはタプルという型がある. タプルとは, 複数の変数を組み合わせたもので, 例えば変数 x, y をひとまとめたにした

$$\begin{pmatrix} x \\ y \end{pmatrix} \quad (4.11)$$

*⁷ 関数の型に出てくる \rightarrow は 2 引数をとる型コンストラクタである. 型コンストラクタに関しては第 XYZ 章で詳しく述べる. 例えば $a \rightarrow b$ という型は $(\rightarrow)ab$ の別名であり, 型コンストラクタ (\rightarrow) に引数 a と b を与えたものと読む.

はタプルである．変数 x, y の型は同じでも良いし，異なっても良い．^{*8}

いまタプルを引数に取る関数

$$f \begin{pmatrix} x \\ y \end{pmatrix} = x + y \quad (4.12)$$

があったとしよう．Haskell にはタプルをとる関数をカーリー化する関数 `curry` があり，

$$(\text{curry } f)xy \quad (4.13)$$

は $x + y$ になる．

逆に，カーリー化された関数

$$f'xy = x + y \quad (4.14)$$

に関しては

$$(\text{uncurry } f') \begin{pmatrix} x \\ y \end{pmatrix} \quad (4.15)$$

のようにアンカーリー化することで，タプルに適用することができる．

* * *

タプルの中身の個数は 0 個または 2 個以上でなければならない，上限は処理系によって定められている．2 個の変数からなるタプルを特別にペア，3 個の変数からなるタプルを特別にトリプルと呼ぶ．中身が 0 個のタプルすなわち `()` は特別に ユニット と呼ぶ．^{*9}

4.3 多相型と型クラス*

整数型 (`Int`) と浮動小数点型 (`Float`) はよく似ている．どちらも値同士を比較可能で，それ故どちらにも等値演算子が定義されている．

^{*8} Haskell では $\begin{pmatrix} x \\ y \end{pmatrix}$ を `(x, y)` と書く．

^{*9} GHC v8.2.1 は最大 62 個の変数からなるタプルまで生成できる．

整数型の等値演算子は

$$(\equiv) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \quad (4.16)$$

であり、浮動小数点型の等値演算子は

$$(\equiv) :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Bool} \quad (4.17)$$

である。

このように型が異なっても（だいたい）同じ意味で定義されている演算子のことを多相的な演算子と呼ぶ。等値演算子は多相的な演算子の例である。

具体的な型を指定せずに、仮の変数で表したものを型パラメタと呼ぶ。我々は型パラメタをボールド体で表す。いま型を表す仮の変数を **a** とし、等値演算子の型を

$$(\equiv) :: \mathbf{a} \rightarrow \mathbf{a} \rightarrow \text{Bool} \quad (4.18)$$

と表現してみよう。このような型パラメタを用いた型を総称して多相型と呼ぶ。

実は式 (4.18) は不完全なものである。このままでは型 **a** に何の制約もないため、等値演算の定義されていない型が来るかもしれないからである。そこで、型自身が所属する、より大きな型があるとしよう。そのような型を我々は型クラスと呼ぶ。例えば型 **Bool**, **Int**, **Integer**, **Float**, **Double** は全て等値演算が定義できるので、型クラス **Eq** に属すとする。この関係を我々は

$$\mathbf{Eq} \supset \text{Bool}, \text{Int}, \text{Integer}, \text{Float}, \text{Double} \quad (4.19)$$

と書く。ここに **Eq** \supset **a** と書いて「型 **a** は型クラス **Eq** のインスタンスである」と読む。

式 (4.18) に型クラスの制約を加えてみよう。型 **a** は型クラス **Eq** に属さなければならないから、新たな記号 \Rightarrow を使って

$$(\equiv) :: \mathbf{Eq} \supset \mathbf{a} \Rightarrow \mathbf{a} \rightarrow \mathbf{a} \rightarrow \text{Bool} \quad (4.20)$$

と書くことにする。^{*10}

型 **a** の変数同士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 **Eq** に属する場合、その型は型クラス **Ord** にも属する。型クラス **Ord** に属する型は比較演算子 $<, \leq, \geq, >$ を提供する。例えば型 **Int** は型クラス **Ord** に属すが、型 **Bool** は型クラス **Ord** に属さない。

型 **a** の変数同士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 **Eq** に属する場合、その型は型クラス **Num** にも属する。型クラス **Num** に属する型は二項演算子 $+, -, *, /$ を提供する。ここに $-$ は二項演算子のマイナスである。

型 **a** が型クラス **Ord** 及び型クラス **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 **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 **Enum** に属する。

型 **a** が型クラス **Real** 及び型クラス **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 **Integral** にも属する。

* * *

便利な型変換演算子をひとつ紹介しておこう。型変換演算子 $\triangle_{\text{Integral}}$ は

$$\triangle_{\text{Integral}} :: \text{Integral} \supset \mathbf{a} \Rightarrow \mathbf{a} \rightarrow \mathbf{b} \quad (4.21)$$

という型を持ち、**Integral** 型クラスの型の変数を、任意の型へ変換する。例えば

$$\left\| \begin{array}{l} x :: \text{Int} \\ x = 1 \end{array} \right. \quad (4.22)$$

$$\left\| \begin{array}{l} y :: \text{Double} \\ y = \triangle_{\text{Integral}} x \end{array} \right. \quad (4.23)$$

^{*10} Haskell では $(==) :: \text{Eq } \mathbf{a} \Rightarrow \mathbf{a} \rightarrow \mathbf{a} \rightarrow \text{Bool}$ と書く。記号 \supset は省略する。

とすることで、**Int** 型の変数 x の値を **Double** 型の変数 y へ代入することができる。^{*11}

* * *

TK: 種

4.4 余談：モノイド

整数全てからなる集合を \mathbb{Z} で表すことにする．計算機科学で整数と言うと、本当の整数と、例えば -2^{63} から $2^{63} - 1$ までの間の整数の意味と両方あるが、今は前者の意味である．

集合 \mathbb{Z} の任意の元 (要素) z を

$$z :: \mathbb{Z} \tag{4.24}$$

と書く．

二つの整数 $z_1, z_2 :: \mathbb{Z}$ があるとしよう．両者の間には足し算 (+) が定義されており、その結果すなわち和もまた整数である．ここで

$$z_1 + z_2 :: \mathbb{Z} \tag{4.25}$$

であるとき、演算子 + が集合 \mathbb{Z} に対して全域性を持つと言う．一般に集合 \mathbb{A} の元に対して二項演算子 \star が定義されていて、 $a_1, a_2 :: \mathbb{A}$ のときに

$$a_1 \star a_2 :: \mathbb{A} \tag{4.26}$$

^{*11} Haskell では

```
x :: Int
x = 1
y :: Double
y = fromIntegral x
```

と書く．

である場合、つまり演算子 \star が集合 A に対して全域性を持つ場合、組み合わせ (A, \star) をマグマと呼ぶ。組み合わせ $(\mathbb{Z}, +)$ はマグマの例であり、 $(\mathbb{Z}, *)$ もマグマの例である。

他に論理集合 $\mathbb{B} = \{\text{True}, \text{False}\}$ に対して、論理和 (\vee) は全域性を持つから、組み合わせ (\mathbb{B}, \vee) はマグマであるし、同様に論理積 (\wedge) も全域性を持つから、組み合わせ (\mathbb{B}, \wedge) もマグマである。論理集合 \mathbb{B} とは論理型 **Bool** を数学風に言い換えたものである。

マグマのうち、演算を2回続ける場合、その順序によって結果が異ならない、つまり

$$(a_1 \star a_2) \star a_3 = a_1 \star (a_2 \star a_3) \quad (4.27)$$

ただし $a_1, a_2, a_3 :: A$ のとき、組み合わせ (A, \star) のことを半群と呼ぶ。この式 (4.27) で表される性質を結合性と呼ぶ。組み合わせ $(\mathbb{Z}, +)$, $(\mathbb{Z}, *)$, (\mathbb{B}, \vee) , (\mathbb{B}, \wedge) はすべて半群である。

ところで、整数全体の集合 \mathbb{Z} には特別な元 $0 :: \mathbb{Z}$ がある。この元 0 は $z :: \mathbb{Z}$ のとき

$$0 + z = z + 0 = z \quad (4.28)$$

という性質を持つ。この 0 を演算 $+$ における単位元と呼ぶ。足し算のことを加法とも言うので 0 のことは加法単位元と呼ぶこともあるし、文字通り零元と呼ぶこともある。

一般に、 $a, \emptyset_{\text{left}}, \emptyset_{\text{right}} :: A$ として

$$\emptyset_{\text{left}} + a = a + \emptyset_{\text{right}} = a \quad (4.29)$$

であるとき、元 $\emptyset_{\text{left}}, \emptyset_{\text{right}}$ を単位元と呼ぶ。我々は多くの場合 $\emptyset_{\text{left}} = \emptyset_{\text{right}}$ であるケースを扱うので、二つの単位元を区別する必要はほとんどないが、必要な場合は \emptyset_{left} を左単位元、 \emptyset_{right} を右単位元と呼ぶ。

組み合わせ $(A, +, \emptyset_{\text{left}}, \emptyset_{\text{right}})$ のことをモノイドまたは単位的半群と呼ぶ。例えば $(\mathbb{Z}, +, 0, 0)$ はモノイドであるし、 $(\mathbb{Z}, *, 1, 1)$, $(\mathbb{B}, \vee, \text{False}, \text{False})$,

表 4.1 モノイド (単位的半群)

型	演算子	単位元
Bool	\vee	False
Bool	\wedge	True
Int	$+$	0
Int	$*$	1
Float	$+$	0
Float	$*$	1

$(\mathbb{B}, \wedge, \text{True}, \text{True})$ もモノイドである. これらのモノイドは全て左単位元と右単位元が同じなので, それぞれ $(\mathbb{Z}, +, 0)$, $(\mathbb{Z}, *, 1)$, $(\mathbb{B}, \vee, \text{False})$, $(\mathbb{B}, \wedge, \text{True})$ とも書く.

このように, 数学者は数の性質を抽象化し, 集合とその集合に対する演算というものの見方をよく行う. プログラミングの言葉で言えば, 複数のクラスに共通のインタフェースを定義するようなものである.

表 4.1 に型と対応するモノイドの単位元, 演算子の一覧を示す.

組み合わせ $(\text{Int}, +, 0)$ はモノイドである. 同様に $(\text{Int}, *, 1)$, $(\text{Float}, +, 0)$, $(\text{Float}, *, 1)$, $(\text{Bool}, \vee, \text{False})$, $(\text{Bool}, \wedge, \text{True})$ もモノイドである.

4.5 この章のまとめ*

1. 集合 \mathbb{A} の元 $a_1, a_2 :: \mathbb{A}$ について, 二項演算子 \star があり $a_1 \star a_2 :: \mathbb{A}$ であるとき, すなわち演算子が全域性を有する場合 (\mathbb{A}, \star) のことをマグマと呼ぶ.
2. マグマ \mathbb{A} の元 $a_1, a_2, a_3 :: \mathbb{A}$ について $(a_1 \star a_2) \star a_3 = a_1 \star (a_2 \star a_3)$ である場合, すなわち演算子が結合性を有する

場合 A を半群と呼ぶ.

3. 半群のうち $\emptyset :: A$ なる元 \emptyset があり, 任意の $a :: A$ に対して $\emptyset \star a = a \star \emptyset = a$ である場合, すなわち単位元が存在する場合 A をモノイドと呼ぶ.
4. 関数は集合 A から集合 A' への写像という型を持つ.
5. 複数引数を取る関数はカーリー化によって, 1 引数をとる複数の関数へ分解される.
6. 型のインタフェースをまとめたものを型クラスと呼ぶ.

第 5 章

リスト

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。

5.1 リスト

同じ型の値を一列に並べたもの、つまりホモニアスな配列のことを リスト と呼ぶ。Python ではリスト `ls` を

```
ls = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python

のように定義できる。

我々も 0 から始まり 9 まで続く整数のリストを `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` と書くことにしよう。ただし、これでは冗長なので 等差数列 に限って簡略化した書き方を許す。例えば 0 から 9 までのリストは `[0, 1...9]` と書いても良い。^{*1}

^{*1} Haskell では `[0, 1..9]` と書く。ピリオドの数に注意しよう。

リストの中身の一つ一つの値のことを要素と呼ぶ。要素のことは元と呼んでも良いが、本書では要素と呼ぶことにする。要素も元も英語の element の和訳である。

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。

今後、リストを指す変数は、リストであることを忘れないように変数名に s をつけて

$$x_s = [0, 1 \dots 9] \quad (5.1)$$

のように書くことにしよう。なお、変数 x とリスト変数 x_s は異なる変数であるとする。^{*2}

* * *

Python ではリスト内包表記が使える。例えば 0 から 9 までの倍数のリストは次のように作った。

```
ls = [x*2 for x in range(0, 10)]
```

Python

ここに `range(a, b)` は a から増加する方向に連続する b 個の整数からなるリストを返す Python の関数である。

我々も内包表記を

$$x_s = [x * 2 \mid x \in [0, 1 \dots 9]] \quad (5.2)$$

のように書こう。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。^{*3}

内包表記の式は複数あっても良い。例えば

$$x_s = [x + y \mid x \in [0, 1 \dots 9], y \in [0, 1 \dots 5], x + y > 3] \quad (5.3)$$

^{*2} Haskell では s を変数名にくっつけて $x_s = [0, 1 \dots 9]$ のように書く習慣がある。

^{*3} Haskell では $x_s = [x * 2 \mid x \in [0, 1 \dots 9]]$ を $x_s = [x * 2 \mid x \leftarrow [0, 1 \dots 9]]$ と書く。

は $0 \leq x \leq 9$ 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。これは Python でいう

```
ls = [\mXVar+\mYVar for x in range(0, 10) for y in range(0, 6) \
      if \mXVar+\mYVar > 3]
```

Python

のことである。^{*4}

整数型 (**Int**) のリストは **[Int]** と書き、整数のリスト型と呼ぶ。一般に **a** 型のリストを **[a]** と書く。仮の型である **a** の事を 型パラメタと呼ぶ。

型 **a** から型 **[a]** を生成する演算子をリスト型コンストラクタと呼んで **List** と書き

$$[a] = \text{List } a \quad (5.4)$$

とする。^{*5}この等式の両辺は変数ではなく型名であることに注意しよう。型コンストラクタの概念は Python には無い（必要無い）が、静的型付け言語である C++ の「クラステンプレート」が相当する。^{*6}

$[x]$ のように **a** 型の変数 x を入れた **[a]** 型の変数を作る演算子をリスト値コンストラクタと呼ぶ。**[a]** 型の変数のことをリスト変数とも呼ぶ。**a** 型の変数 x からリスト値コンストラクタを使ってリスト x_s を作ることは

$$x_s = [x] \quad (5.5)$$

と書く。^{*7}

リスト型を表す **[a]** と、1 要素のリストである $[x]$ の違いにはいつも気をつけておこう。本書では中身がボールドローマン体ならばリスト型、中身がイタリック体ならリスト値である。

^{*4} Haskell では $xs = [x+y \mid x \leftarrow [0,1..9], y \leftarrow [0,1..5], x+y > 3]$ と書く。

^{*5} Haskell ではこの式の右辺を $[] \ a$ とも書ける。

^{*6} Haskell では表記上コンテナ型 **[a]** と型コンストラクタ式 **List a** を区別せず、両者とも **[a]** と書く。

^{*7} Haskell では $xs = [x]$ と書く。

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。コンテナ型の変数をコンテナ変数と呼ぶ。コンテナ型は多相型の一種である。

* * *

リストは結合できる。例えばリスト x_s とリスト y_s を結合したリストは

$$x_s \oplus y_s \tag{5.6}$$

と表現する。リストの結合演算子の型は

$$(\oplus) :: [a] \rightarrow [a] \rightarrow [a] \tag{5.7}$$

である。^{*8}

* * *

リストは空でもよい。空リストは $[]$ で表す。^{*9}

関数 `null` はリストが空リストかどうかを判定する。リスト x_s が空リストの場合

$$\text{null } x_s \tag{5.8}$$

は `True` を、そうでなければ `False` を返す。

* * *

我々は無限リストを持つことができる。例えば自然数を表すリスト n_s は

$$n_s = [1, 2 \dots]$$

と書くことができる。^{*10}

^{*8} Haskell では $x_s \oplus y_s$ を `xs++ys` と書く。

^{*9} Haskell では空リストを $[]$ で表す。

^{*10} Haskell では $n_s = [1, 2 \dots]$ を `ns = [1, 2..]` と書く。

無限リストを扱えるのは、我々がいつも遅延評価を行うからである。遅延評価とは、本当の計算は必要になるまで行わないという方式のことである。

もし本当に無限リストを計算機の上で再現する必要があったなら、計算機には無限のメモリが必要になってしまう。しかし我々は、計算が必要になるまで評価を行わないので、無限リストの中から有限個の要素が取り出されるのを待つことができるのである。例えば関数 `take n ns` はリスト `ns` から最初の `n` 個の要素からなるリストを返す。いま

$$x_s = \text{take } 5 \, ns$$

とすると、リスト `x_s` は `x_s = [1, 2 \cdots 5]` という値を持つ。^{*11}

関数 `take` の型は

$$\text{take} :: \text{Int} \rightarrow [a] \rightarrow [a] \quad (5.9)$$

である。

* * *

リスト `x_s` の `n` 番目の要素には

$$x_s !! n \quad (5.10)$$

とすることでアクセスできる。^{*12}

5.2 畳み込み

我々はよくリストの総和を表現するために総和演算子 (\sum) を使う。総和演算子とはリスト $[x_0, x_1 \cdots x_n]$ に対して

$$\sum [x_0, x_1 \cdots x_n] = x_0 + x_1 + \cdots + x_n \quad (5.11)$$

^{*11} Haskell では `xs = take 5 ns` と書く。

^{*12} Haskell では `x_s !! n` を `xs !! n` と書く。

で定義される演算子である．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，任意の二項演算子を \star として

$$\bigcup_a^\star [x_0, x_1 \cdots x_n] = a \star x_0 \star x_1 \star \cdots \star x_n \quad (5.12)$$

であると定義する．

この新しい演算子 \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．^{*13}

Python 2.7 には畳み込み演算子に相当する `reduce` 関数があり，リスト `ls` の総和 `s` を

Python

```
# Python 2.7
ls = [0, 1, 2, 3, 4, 5]
s = reduce(lambda x, y: \mXVar+\mYVar, ls, 0)
```

のように求めることができる．この `reduce` 関数は Python バージョン 3 では非推奨になっているが，Ruby には受け継がれていて，Ruby では

Ruby

```
ls = [0, 1, 2, 3, 4, 5]
s = ls.inject(0) { |x, y| \mXVar+\mYVar }
```

と書ける．

リストの総和をとる演算子 \sum は

$$\sum x_s = \bigcup_0^+ x_s \quad (5.13)$$

^{*13} Haskell では $\bigcup_a^\star x_s$ を `foldl (*) a xs` と書く．

とすれば得られる．この式は両辺の x_s を省略して

$$\Sigma = \bigcup_0^+ \quad (5.14)$$

とも書く．このように，ラムダ式を使わずに引数を省略してしまう書き方を ポイントフリースタイル と呼ぶ．ポイントフリースタイルは今後も頻出するので，是非慣れておいてもらいたい．^{*14}

リストの要素のすべての積をとる演算子 Π は

$$\Pi = \bigcup_1^* \quad (5.15)$$

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

$$\bigcup :: (\mathbf{a} \rightarrow \mathbf{b} \rightarrow \mathbf{a}) \rightarrow \mathbf{a} \rightarrow [\mathbf{b}] \rightarrow \mathbf{a} \quad (5.16)$$

である．

* * *

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．

$$\bigcup_a^\star [x_0, x_1 \cdots x_n] = (x_0 \star (x_1 \star \cdots \star (x_n \star a))) \quad (5.17)$$

これは 右畳み込み と呼ばれる演算子である．^{*15}

* * *

^{*14} Haskell はポイントフリースタイルをサポートする．

^{*15} Haskell では $\bigcup_a^\star x_s$ を `foldr (*) xs a` と書く．引数の順序に注意しよう．

畳み込み演算子の面白い応用例を示そう。リストの結合演算子 (\oplus) を使
うと

$$\bigcup_{[]}^{\oplus} [[0, 1, 2, [3, 4, 5], \dots]] = [0, 1, 2, 3, 4, 5, \dots] \quad (5.18)$$

であるから、演算子 $\bigcup_{[]}^{\oplus}$ 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

$$\flat = \bigcup_{[]}^{\oplus} \quad (5.19)$$

と定義することにする。^{*16}

5.3 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。Python ではリ
スト `ls` に関数 `f` を適用するときには

Python

```
map(f, ls)
```

のように `map` 関数を用いる。例えば

Python

```
f = lambda x: 100+x
ls = [1, 2, 3, 4, 5]
ms = map(f, ls)
```

とすると、結果として `ms` には `[101, 102, 103, 104, 105]` が入る。

^{*16} Haskell では演算子 \flat の代わりに `concat` 関数（または `join` 関数）を使う。

このように引数として関数 f とリスト $[x_0, x_1 \cdots x_n]$ を取り、戻り値として $[f x_0, f x_1 \cdots f x_n]$ を返す演算子 \odot を考えよう。このとき

$$f \odot [x_0, x_1 \cdots x_n] = [f x_0, f x_1 \cdots f x_n] \quad (5.20)$$

であると定義する。この演算子 \odot をリストの マップ演算子 と呼ぶ。^{*17}

リストのマップ演算子の型は

$$\odot :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \quad (5.21)$$

である。矢印 \rightarrow は右結合なので、これは

$$\odot :: (a \rightarrow b) \rightarrow ([a] \rightarrow [b]) \quad (5.22)$$

の意味でもある。念のため上式に注釈を加えると

$$\odot :: \underbrace{(a \rightarrow b)}_f \rightarrow \left(\underbrace{[a]}_{[x_0, x_1 \cdots x_n]} \rightarrow \underbrace{[b]}_{[f x_0, f x_1 \cdots f x_n]} \right) \quad (5.23)$$

である。

ここで f と $f \odot$ の型を並べてみると

$$f :: a \rightarrow b \quad (5.24)$$

$$f \odot :: [a] \rightarrow [b] \quad (5.25)$$

となり、マップ演算子は何をしているのか一目瞭然になる。

TK:liftM

具体例を見てみよう。先程の Python コードの例にあわせて

$$f = \lambda x \mapsto 100 + x \quad (5.26)$$

$$x_s = [1, 2, 3, 4, 5] \quad (5.27)$$

$$y_s = f \odot x_s \quad (5.28)$$

とすると y_s の値は $[101, 102, 103, 104, 105]$ となる。

^{*17} Haskell では $f \odot x_s$ を `map f xs` または `f <$> xs` と書く。ただし演算子 `<$>` は `fmap` 演算子の中置バージョンである。

5.4 余談：リストの実装

ここでリストの実装について述べておこう．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．我々はリストを 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変数 `first` と変数 `rest` からなるペアの集合である．変数 `first` がリストの要素を参照し，変数 `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特別な値を持つ．

リストのための特別な表現

$$\text{first} : \text{rest} \quad (5.29)$$

を用い，リファレンス `first` はリストが保持する型，リファレンス `rest` はリスト型であるとする．演算子：を LISP に倣って cons 演算子 と呼ぶ．^{*18}

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．

$$[x_0, x_1, x_2 \cdots x_n] = x_0 : [x_1, x_2 \cdots x_n] \quad (5.30)$$

$$= x_0 : x_1 : [x_2 \cdots x_n] \quad (5.31)$$

$$= x_0 : x_1 : x_2 : \cdots : x_n : [] \quad (5.32)$$

cons 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．

マップ演算子の実装は，リストの実装に踏み込めば簡単である．空でないリストは必ず $x : x_s$ へと分解できるから

$$\begin{cases} f \odot [] = [] \\ f \odot (x : x_s) = (f x) : (f \odot x_s) \end{cases} \quad (5.33)$$

^{*18} Haskell でも要素 x をリスト xs の先頭に追加することを $x:xs$ と書く．

とマップ演算子 (\odot) を定義できる。つまりマップ演算子 (\odot) は cons 演算子 ($:$) から作ることができる。換言すれば、マップ演算子はシンタックスシュガーである。

Haskell では任意のリスト x_s に対し、次の関数が用意されている。

$\text{head } x_s \dots x_s$ の先頭要素

$\text{tail } x_s \dots x_s$ の 2 番目以降の要素からなるリスト

これらは LISP の `car` 関数、`cdr` 関数と同じものであり、この二者を用いればどのようなリストの処理も可能である。

このように基本的な関数から高機能な関数を実装する方法はよく行われる。この例では cons 演算子からマップ演算子を合成した。

* * *

リストを引数にとる関数はいつでも

$$f(x : x_s) = \dots \quad (5.34)$$

という風にパターンマッチを行えるが、式の右辺でリスト全体すなわち $(x : x_s)$ を参照したい場合もあるであろう。そのような場合は

$$f a_s @ (x : x_s) = \dots \quad (5.35)$$

として、変数 a_s でリスト全体を参照することも可能である。このような記法を as パタン と呼ぶ。^{*19}

5.5 この章のまとめ

1. 配列はリストである。配列は $[x_0, x_1, x_2]$ のように表記する。
2. 等差数列の配列は $[0, 1 \dots 10]$ のように途中を \dots で省略できる。

^{*19} Haskell では $f a_s @ (x : x_s)$ を `f as@(x:xs)` と書く。

3. 配列は無限長であってもよい。無限長配列は $[0, 1 \dots]$ のように表記する。
4. 1 要素のリストは $[x]$ のように表記する。この括弧をリスト値コンストラクタと呼ぶ。
5. 空リストは $[]$ と表す。
6. \mathbf{a} 型のリストを $[\mathbf{a}]$ 型で表す。
7. リスト型コンストラクタ **List** は型パラメタ \mathbf{a} に作用して $[\mathbf{a}]$ を生成する。
8. リストは先頭要素と続くリストから定義される。先頭要素を x とし、続くリストを x_s とすると $x : x_s$ はリストである。ここに演算子 $:$ は結合 (cons) 演算子である。
9. リストはリスト結合 (append) できる。リスト y_s とリスト z_s のリスト結合は $y_s \oplus z_s$ である。
10. 左畳み込み演算子 $\bigcup_a^\star [x_0, x_1 \dots x_n]$ は $a \star x_0 \star x_1 \star \dots \star x_n$ を返す。
11. 右畳み込み演算子 $\bigcup_a^\star [x_0, x_1 \dots x_n]$ は $(x_0 \star (x_1 \star \dots \star (x_n \star a)))$ を返す。
12. 平坦化演算子 $\flat = \bigcup_{[]}^\oplus$ はリストを平坦化する。
13. 関数 $f :: \mathbf{a} \rightarrow \mathbf{a}$ はマップ演算子 \odot を用いて $f \odot [x_0, x_1 \dots x_n] = [f x_0, f x_1 \dots f x_n]$ のようにリスト $[x_0, x_1 \dots x_n]$ に適用できる。
14. マップ演算子 \odot の型は $(\mathbf{a} \rightarrow \mathbf{b}) \rightarrow \mathbf{a} \rightarrow \mathbf{b}$ である。
15. $\text{head}(x : x_s) = x, \text{tail}(x : x_s) = x_s$ である。

第 6 章

再帰

~~TK: To be written.~~

6.1 関数の再帰適用

関数は内部で自分自身を適用しても良い。例えば x の階乗 ($x!$) を返す関数 `fact` は

$$\text{fact } x = \text{case } x \text{ of } \begin{cases} 0 & \rightarrow 1 \\ - & \rightarrow x * \text{fact}(x - 1) \end{cases} \quad (6.1)$$

と定義できる。関数が自分自身を適用することを関数の再帰適用と呼ぶ。^{*1}

これで我々は関数の適用、変数の代入、ラムダ式、条件式、再帰の方法を学んだわけである。これだけあれば、原理的にはどのようなアルゴリズムも書くことができる。今日からはカーリー風な数学であらゆるアルゴリズムを表

^{*1} Haskell では

```
fact x = case x of
  0 -> 1
  _ -> x * fact (x - 1)
```

と書く。

現できるのである！

* * *

cons 演算子 ($:$) は関数引数のパターンにも使える。これは、例えばリストの和をとる関数 `sum` は

$$\begin{cases} \text{sum} [] = 0 \\ \text{sum}(x : x_s) = x + \text{sum } x_s \end{cases} \quad (6.2)$$

のようにも定義できるということである。^{*2}

なお、関数は再帰させるたびに計算機のスタックメモリを消費する。これを回避するためのテクニックが、次節で述べる末尾再帰である。

6.2 末尾再帰

計算機科学者は、同じ再帰でも末尾再帰という再帰のスタイルを好む。末尾再帰とは、関数の再帰適用を関数定義の末尾にすることである。この章に出てきた階乗関数 `fact` を例にとろう。階乗関数 `fact` は

$$\text{fact } x = \text{case } x \text{ of } \begin{cases} 0 & \rightarrow 1 \\ - & \rightarrow x * \text{fact}(x - 1) \end{cases} \quad (6.3)$$

のような形をしていた。末尾の関数をよりはっきりさせるために演算子 ($*$) を前置にして

$$\text{fact } x = \text{case } x \text{ of } \begin{cases} 0 & \rightarrow 1 \\ - & \rightarrow (*)x(\text{fact}(x - 1)) \end{cases} \quad (6.4)$$

^{*2} Haskell では

```
sum []      = 0
sum (x:xs) = x + sum xs
```

と書く。

と書いてみよう．この定義の末尾の式は

$$(*)x(\text{fact}(x-1)) \quad (6.5)$$

である．これだと末尾の関数は `fact` ではなく演算子 `(*)` なので，末尾に再帰適用を行ったことにはならない．

そこで，次のように形を変えた階乗関数 `fact'` を考えてみる．

$$\text{fact}' ax = \text{case } x \text{ of } \begin{cases} 0 & \rightarrow 1 \\ - & \rightarrow \text{fact}'(a * x)(x-1) \end{cases} \quad (6.6)$$

こうすれば末尾の関数がもとの `fact'` と一致する．^{*3}

関数 `fact` と違い関数 `fact'` は引数を 2 個とる．関数 `fact'` を使って `x` の階乗を求める場合は `fact' 1 x` と第 1 引数に 1 を与えることにする．この第 1 引数 `a` はアキュムレータという．アキュムレータが演算の途中経過を引き渡していくイメージを描けば，末尾再帰の意味が理解できるだろう．

計算機科学者が末尾再帰を好む理由は，Haskell を含む幾つかのプログラミング言語処理系が末尾再帰最適化を行うからである．末尾再帰最適化とは，一言で言うと再帰を計算機が扱いやすいループに置き換えることである．では最初から我々もループで関数を表現しておけば，と思われるかもしれないが，再帰以外の方法でループを表現する場合には必ず変数（ループカウンタ）への破壊的代入が必要になるため，我々は末尾再帰に慎ましくループを隠すのである．

^{*3} Haskell では

```
fact a x = case x of 0 -> 1
                  _ -> fact' (a*x) (x-1)
```

と書く．

6.3 遅延評価

Haskell は、意図しない限り遅延評価を行う。これは特に左畳み込み演算子 (\bigcup) を使う場合に問題となる。いま $x_s = [x_0, x_1, x_2, x_3]$ とすると、左畳み込み演算 $\bigcup_0^+ x_s$ は

$$\bigcup_0^+ x_s = \bigcup_0^+ (x_0 : x_1 : x_2 : []) \quad (6.7)$$

$$= \bigcup_{0+x_0}^+ (x_1 : x_2 : x_3 : []) \quad (6.8)$$

$$= \bigcup_{(0+x_0)+x_1}^+ (x_2 : x_3 : []) \quad (6.9)$$

$$= \bigcup_{((0+x_0)+x_1)+x_2}^+ (x_3 : []) \quad (6.10)$$

$$= \bigcup_{(((0+x_0)+x_1)+x_2)+x_3}^+ [] \quad (6.11)$$

$$= (((0+x_0)+x_1)+x_2)+x_3 \quad (6.12)$$

と展開される。遅延評価のために、Haskell 処理系は値ではなく式をメモリにストアしなければならないが、左畳み込み演算は大きなメモリを必要としがちである。もし例えば予め $0+x_0$ を先に計算しておくなど左畳み込みだけ先に評価しておけば、大いにメモリの節約になる。そのために Haskell は「遅延評価無し」の左畳み込み演算子を用意している。^{*4}

^{*4} 「遅延評価無し」の左畳み込み演算子を Haskell では `old1` と書く。

6.4 余談：クロージャ

ラムダ式をサポートするほとんどのプログラミング言語は、レキシカルクロージャをサポートする。レキシカルクロージャとは、ラムダ式が定義された時点での、周囲の環境をラムダ式に埋め込む機構である。例えば

$$a = 100 \tag{6.13}$$

$$h = a + \diamond \tag{6.14}$$

というラムダ式があるとする。当然我々は関数 h がいつも $h = 100 + \diamond$ であることを期待するし、Haskell においてはいつも保証される。^{*5}

ところが、参照透過性のない言語、言い換えると変数への破壊的代入が許されている言語では、変数 a の値がいつ変わっても不思議ではない。そこで、それらの言語では関数 h が定義された時点での a の値を、関数 h の定義に含めておく。これがレキシカルクロージャの考え方である。

Haskell ではそもそも変数への破壊的代入がないので、関数 h がレキシカルクロージャであるかどうか悩む必要はない。あえて言えば、Haskell ではラムダ式はいつもレキシカルクロージャである。もしあなたのそばの C++ プログラマが「え？ Haskell にはレキシカルクロージャが無いの？」などと聞いてきたら、「ええ、Haskell には破壊的代入すらありませんから」と答えておこう。

^{*5} Haskell では

```
a = 100
h = \x -> a+x
```

と書く。

6.5 この章のまとめ

1. 関数は再帰適用できる．ループは再帰適用によって実現する．
2. 末尾再帰はスタックを消費しないように最適化される．
3. 関数はいつも遅延評価される．そのため無限リストを扱うことも可能である．

Haskell によるクイックソート

我々の記法を使うと、クイックソートは次のように定義できる．

$$\left\{ \begin{array}{l} \text{srt} [] = [] \\ \text{srt}(x : x_s) = (\text{srt } a_s) \oplus [x] \oplus (\text{srt } b_s) \\ \text{where } \left\{ \begin{array}{l} a_s \triangleq [a \mid a \in x_s, a \leq x] \\ b_s \triangleq [b \mid b \in x_s, b > x] \end{array} \right. \end{array} \right.$$

Haskell では

```
srt [] = []
srt(x:xs) = srt as ++ [x] ++ srt bs where
  as = [a | a<-xs, a<=x]
  bs = [b | b<-xs, b>x]
```

Haskell

と書く．このコードはしばしば Haskell のパワーを示すために紹介される．しかし、クイックソートのピボットとして常にリストの先頭要素を用いているため、必ずしも良いコードではない．

第 7 章

Maybe*

この章では計算結果が正しいかもしれないし、正しくないかもしれないという曖昧な状況を表す型を導入する。手始めに Python でクラス `Possibly` を実装し、それがカーリー風の数式で綺麗に書けることを示す。またリストとの共通点についても見ていくことにする。

7.1 Possibly

計算の途中で、計算にまつわる状態を残りの計算に引き継ぎたくなる場合がある。例えば、整数 x, y, z があり $x = y/z$ なる値を続く計算で利用したいとする。だが $z \equiv 0$ のときには x は正しく計算されない。こんなときプログラマが取れる手段は

- $x = y/z$ を計算した時点で**ゼロ除算例外**を発生させ、プログラムの制御を他の場所へ移す（大域ジャンプを行う）
- グローバル変数にゼロ除算エラーが起こったことを記録しておき、 x にはとりあえずの数値、例えば 0 を代入しておいて、計算を続行させる
- x にエラー状態を示す印を新たにつけておいて、計算を続行させる

といったところだろう。

大域ジャンプも、グローバル変数の書き換えも破壊的代入を伴うものであり、受け入れがたい。そこで我々は第三のエラー状態を示す印をつける方法を採用することにする。普通変数が整数だろうが実数だろうが、計算機表現には余分なビットが残っていないので、変数をラップする次のようなクラス `Possibly` を導入することにしよう。メンバ変数 `value` が値を、メンバ変数 `valid` がエラーの有無を表す。

```
class Possibly:
    def __init__(self, a_valid, a_value = 0):
        self.valid = a_valid
        self.value = a_value
```

Python

例えば整数値 123 を持つ `Possibly` クラスの値 `p` は

```
p = Possibly(True, 123)
```

Python

として生成できるし、`Possibly` 値 `p` が計算エラーを表す場合は

```
p = Possibly(False)
```

Python

と初期化できる。

ここで、引数に 1 を加えて返す関数 `f` があるとしよう。関数 `f` の定義は次の通りである。

```
f = lambda x: 1+x
```

Python

関数 `f` に直接 `Possibly` 値 `p` を食わせるとランタイムエラーを引き起こす。

Python

```
q = f(p) # エラー!!
```

これは関数 `f` が引数として数値を期待していたにもかかわらず、Possibly クラスの値が渡されたからである。もし関数 `f` のほうをいじりたくないとなれば、次のような関数 `map_over` を使って

Python

```
q = map_over(f, p)
```

というふうに間接的に関数適用を行う必要がある。

関数 `map_over(f, p)` はもし `p` がエラーを表す値でなければ中身の値を関数 `f` に適用し、その結果を Possibly クラスに包んで返す。もし `p` がエラー値を表す値であれば、結果もエラー値である。関数 `map_over` の実装は次のようになる。

Python

```
def map_over(f, p):  
    if p.valid == True:  
        return Possibly(True, f(p.value))  
    else:  
        return Possibly(False)
```

さて、次節では以上のようなことを抽象数学的に綺麗に描いてみよう。

7.2 Maybe

もう一度振り出しに戻る。

整数 x, y, z があり $x = y/z$ という式があるとする。この式は $z \equiv 0$ のときにはゼロ除算エラーである。しかし「例外」は内部状態の書き換えであり、我々の計算に入れたくない。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 $u?$ のように?をつけた変数に入れて、忘れないようにしておこう。

ここで変数 $u_?$ が取り得る値は正しく計算された値 x をラップしたものか、あるいはエラーを表す値 \emptyset である。このように計算結果に「意味付け」をすることを文脈に入れると言う。定数 \emptyset は「ナッシング」と呼ぶ。^{*1}

この変数 $u_?$ はもはや整数 (**Int**) 型とは言えない。そこでこの $u_?$ の型を $\text{Maybe}[\text{Int}]$ と表して「Maybe 整数 (おそらく整数)」型と呼ぶことにしよう。型 a から型 $\text{Maybe}[a]$ を生成するには型コンストラクタ **Maybe** を用いて

$$\text{Maybe}[a] = \text{Maybe } a \quad (7.1)$$

とする。^{*2}

a 型の変数を $\text{Maybe}[a]$ 型の変数に代入するには、次の値コンストラクタ $\text{Just}[\dots]$ を用いて

$$u_? = \text{Just}[x] \quad (7.2)$$

と書く。^{*3}

変数 x が一度ゼロ除算の危険性に「汚染」された場合、その後ずっと Maybe 変数に入れ続けなければいけない。そこで、普通の変数を引数にとる関数 f に Maybe 変数 $u_?$ を食わせるには、リストの時と同じようなマップ演算子が必要になる。具体的には、変数 x が **Int** 型として、Maybe 変数 $u_? = \text{Just}[x]$ が与えられたとき

$$f \sqcap u_? = \text{Just}[fx] \quad (7.3)$$

となるような Maybe バージョンのマップ演算子 \sqcap を用いる。ここに $f \sqcap u_?$ の型は、もし $f :: \text{Int} \rightarrow \text{Float}$ ならば $\text{Maybe}[\text{Float}]$ である。

^{*1} Haskell では \emptyset を **Nothing** と書く。

^{*2} Haskell では表記上コンテナ型 $\text{Maybe}[a]$ と型コンストラクタ式 **Maybe** a を区別せず、両者とも **Maybe** a と書く。

^{*3} Haskell では $u_? = \text{Just}[x]$ を $u = \text{Just } x$ と書く。Maybe を表す疑問符は省略する。

実際には $u? \equiv \emptyset$ の可能性も考えなければならないから、Maybe バージョンのマップ演算子は

$$f \sqcap u? = \text{case } u? \text{ of } \begin{cases} \text{Just } [x] & \dashrightarrow \text{Just } [f x] \\ - & \dashrightarrow \emptyset \end{cases} \quad (7.4)$$

でなければならない。この Maybe バージョンのマップ演算子 \sqcap は

$$\begin{cases} f \sqcap \text{Just } [x] = \text{Just } [f x] \\ f \sqcap \emptyset = \emptyset \end{cases} \quad (7.5)$$

と定義すれば得られる。^{*4}

今後、普通の（引数に Maybe が来ることを想定していない）関数 f を Maybe 型である変数 $u?$ に適用させるときには、必ず

$$v? = f \sqcap u? \quad (7.6)$$

のように Maybe バージョンのマップ演算子 \sqcap を用いることにする。これはプログラムの安全性のためである。変数が一旦ゼロ除算の可能性に汚染されたら、最後まで Maybe に包んでおかねばならない。^{*5}

Python で Maybe の概念を忠実になぞることは難しい。と言うのも Python は動的型付け言語であるため、型コンストラクタという概念が無いからだ。一方で Maybe の概念を静的型付け言語である C++ や Java で実現することはできる。そこで C++ の本物のコードで示しておこう。ただし

^{*4} Haskell では

```
f <$> Just x  = Just (f x)
f <$> Nothing = Nothing
```

と書く。

^{*5} Haskell では Maybe バージョンのマップ演算子に特別な記号、関数名が与えられていない。その代わり第 8 章で述べる一般マップ演算子 \cdot に相当する `fmap` 関数を用い $v = \text{fmap } f \ u$ または $v = f \ <\$> \ u$ のように書く。Haskell は型推論を行なうため変数 u が Maybe であれば Maybe バージョンのマップ演算子（関数）が適用され、もし u がリストであれば通常のマップ関数である `map` が適用される。

ポインタを使わないでおいたので C++ プログラマも Java プログラマも参考にできるだろう。

Maybe は次の maybe クラステンプレートで表現できる。(Java プログラマへの注意：これは maybe<a>クラスの定義と同じ意味である。)

C++

```
template <typename a> class maybe {  
    private:  
        a value;  
        bool valid;  
    public:  
        maybe(): value(0), valid(false) { }  
        maybe(a a_value): value(a_value), valid(true) { }  
        a get_value() const { return value; }  
        bool is_valid() const { return valid; }  
};
```

デフォルトコンストラクタ maybe() は例外的な状況を表す 0 を生成し、1 引数コンストラクタ maybe(a) は maybe<a> で包んだ引数値を生成する。

C++ プログラムで良く見かけるクラス設計と違い、この maybe クラスはコンストラクタ以外に中身を書き換える手段が提供されていない。これが破壊的代入の禁止が意味することである。

当然我々には Maybe バージョンのマップ演算子が必要である。ここでは関数 map_over として書いてみよう。(Java プログラマへの注意：関数 map_over はどのクラスにも属していないが、それで正解なのである。)


```

template <class a, class b, class fn>
maybe<b> map_over(fn f, maybe<a> u) {
    if (u.is_valid()) {
        return maybe<b>(f(u.get_value()));
    }
    else {
        return maybe<b>();
    }
}

```

C++

テンプレートの 2 番目の引数 `fn` は関数 `f` を受け取るために必要である。C++ はコンパイル時までにすべての変数の型が決定していないといけないが、関数 `f` の型は関数 `map_over` 設計時には確定できないため、このようにテンプレートにしている。

整数 x から Maybe 値 $u_? = \text{Just}[x]$ を作り、関数 $g x = 1 + x$ を Maybe 値 $u_?$ に食わせて Maybe 値 $v_?$ ただし

$$v_? = g \sqcap u_? \quad (7.7)$$

を得ることを C++ では次のように書くことになる。

```

int x = 123;
maybe<int> u(x);
auto g = [](int x) -> int { return 1+x; };
maybe<int> v = map_over(g, u);

```

C++

注意してほしいのは `g(x)` も `map_over(g, u)` も正当なコードだが `g(u)` は型エラーであることだ。また `g(u.get_value())` は正当なコードだが、わざわざ `u` が持つ文脈を捨てることになる。

7.3 リストと Maybe*

関数 f を Maybe 値 $u_?$ に適用するために

$$v_? = f \sqcap u_? \quad (7.8)$$

のような Maybe バージョンのマップ演算子 (\boxtimes) を使った. 一方で, 同じ関数 f をリスト x_s に適用するには

$$y_s = f \odot x_s \quad (7.9)$$

のようなリストバージョンのマップ演算子 (\odot) を使った.

リストバージョンのマップ演算子 (\odot) をもし C++ で書くとしたら, 次のようなコードになる. ここでリスト型として C++ の標準テンプレートライブラリ (STL) の `std::list` クラスを流用した.

C++

```
template <class a, class b, class fn>
std::list<b> map_over(fn f, std::list<a> xs) {
    std::list<b> ys(xs.size());
    auto i = xs.cbegin();
    auto j = ys.begin();
    while (i != xs.cend()) {
        *j = f(*i); ++i; ++j;
    }
    return std::list<b>(ys);
}
```

この関数 `map_over` の中身部分はどうでもよい. それよりも, リストバージョンのマップ演算子の C++ 関数のインタフェースと, Maybe バージョンのマップ演算子の C++ 関数のインタフェースを見比べてみよう.

C++

```
// List
template <class a, class b, class fn>
std::list<b> map_over(fn f, std::list<a> xs);

// Maybe
template <class a, class b, class fn>
maybe<b> map_over(fn f, maybe<a> u);
```

やはりそっくりである. であるならば, うまく統一したい. C++ では次のような書き方が文法的には可能である.

C++

```
template <class a, class b, template<class> X, class fn>
X<b> map_over(fn f, X<a> x);
```

これは一見上手く行きそうに見えるが、このコードは `map_over` のインスタンス化で躓くため、次のように `b` 型のダミー変数が必要になる。

C++

```
template <class a, class b, template<class> X, class fn>
X<b> map_over(fn f, X<a> x, b dummy);
```

残念なことに、いずれのコードにしてもリストと `Maybe` の本質的な抽象化にはなっていない。型 `X` がマップ可能なコンテナであることをテンプレート機構を使って保証することができないためである。この問題は C++20 で導入予定の「コンセプト」機能によって解決する見込みである。

一方で、数学者たちが見つけた圏という代数的構造が、リストも `Maybe` も統一的に扱うことを可能にしている。これを発見したのは Eugenio Moggi を始めとする計算機科学者たちである。この人類の英知は第 8 章から見ていくことにしよう。

* * *

~~TK: To be written.~~

$$\sharp \text{Just} [\text{Just} [x]] = \text{Just} [x] \quad (7.10)$$

7.4 余談: Either

`Maybe` とよく似た型に `Either` がある。 `Maybe` が `a` 型または \emptyset のいずれかの値をとったように、 `Either` は `a` 型または `b` 型のいずれかの値を取る。`a` 型または `b` 型を取る `Either` 型の変数 `ei` があるとする、

$$e_i :: \text{Either} [a\ b] \quad (7.11)$$

と書く．Either 型は型 **a** および **b** から型コンストラクタを用いて

$$\text{Either} \llbracket \mathbf{a} \mathbf{b} \rrbracket = \mathbf{Either} \mathbf{a} \mathbf{b} \quad (7.12)$$

のように作られる．*6

Either には値コンストラクタが2種類あり，それぞれ $\text{Right} \llbracket x \rrbracket$ と $\text{Left} \llbracket x \rrbracket$ である．値コンストラクタは

$$e! = \text{Right} \llbracket x \rrbracket \quad (7.13)$$

または

$$e! = \text{Left} \llbracket x \rrbracket \quad (7.14)$$

のように使う．*7

Either はより複雑な計算エラーが発生する場合に用いる．Maybe が単に失敗を表す \emptyset しか表現できなかったのに対し，Either は任意の型の変数で表現できる．習慣的に，正しい (right) 計算結果は $\text{Right} \llbracket x \rrbracket$ 値コンストラクタで格納し，残された (left) エラーの情報は $\text{Left} \llbracket x \rrbracket$ 値コンストラクタで格納する．

Either 型は C の共有型 (union) や C++ のバリエーション型 (`std::variant`) に近い．

7.5 この章のまとめ

1. ある型 **a** からその Maybe 型 $\text{Maybe} \llbracket \mathbf{a} \rrbracket$ を作ることを $\text{Maybe} \llbracket \mathbf{a} \rrbracket = \mathbf{Maybe} \mathbf{a}$ と書く．ここに **Maybe** は Maybe 型コンストラクタである．
2. ある変数 x から Maybe 変数 $u?$ を作るには $u? = \text{Just} \llbracket x \rrbracket$ とする．ここに $\text{Just} \llbracket \dots \rrbracket$ は Maybe 値コンストラクタである．

*6 Haskell では $\text{Either} \llbracket \mathbf{a} \mathbf{b} \rrbracket$ も **Either a b** も区別せずに **Either a b** と書く．

*7 Haskell ではそれぞれ $e = \text{Right } x$ および $e = \text{Left } x$ と書く．

3. Maybe 変数は $\text{Just} \llbracket x \rrbracket$ のような値か、かまたは \emptyset なる「ナッシング」値のかどちらかを持つことができる。
4. 普通関数 $f :: a \rightarrow b$ を Maybe 値に適用することはできない。関数 f を Maybe 値に適用するには $f \sqcap u?$ のように Maybe マップ演算子が必要であり、この関数適用の結果は $\text{Maybe} \llbracket b \rrbracket$ 型である。
5. Either 変数は二つの型のいずれかを持つことができ、 $e! = \text{Right} \llbracket x \rrbracket$ または $e! = \text{Left} \llbracket x \rrbracket$ のように生成する。

第 8 章

関手*

A...

8.1 圏と関手

\mathbf{a} 型の変数 $x, y :: \mathbf{a}$ について, 関数 $f :: \mathbf{a} \rightarrow \mathbf{a}$ があり

$$y = fx \quad (8.1)$$

であるとして. このように型 \mathbf{a} で閉じた世界を仮に \mathbf{a} 世界と呼ぶことにする.

型 $\text{Maybe}[\![\mathbf{a}]\!]$ の変数 $u?, v? :: \text{Maybe}[\![\mathbf{a}]\!]$ について, 関数

$$g :: \text{Maybe}[\![\mathbf{a}]\!] \rightarrow \text{Maybe}[\![\mathbf{a}]\!] \quad (8.2)$$

があり

$$v? = gu? \quad (8.3)$$

であるとして. このように $\text{Maybe}[\![\mathbf{a}]\!]$ で閉じた世界を仮に $\text{Maybe}[\![\mathbf{a}]\!]$ 世界と呼ぶことにする.

ここで、変数 x, y と Maybe 変数 $u?, v?$ は Maybe 値コンストラクタによって

$$u? = \text{Just} \llbracket x \rrbracket \quad (8.4)$$

$$v? = \text{Just} \llbracket y \rrbracket \quad (8.5)$$

の関係にあるとしよう。値コンストラクタは値を \mathbf{a} 世界から $\text{Maybe} \llbracket \mathbf{a} \rrbracket$ 世界へとジャンプさせる機能を持っている。

他に \mathbf{a} 世界から $\text{Maybe} \llbracket \mathbf{a} \rrbracket$ 世界へジャンプさせるものがあるだろうか。よく考えてみると、マップ演算子もそうである。いま $u? = \text{Just} \llbracket x \rrbracket, v? = \text{Just} \llbracket y \rrbracket$ なのだから、 \mathbf{a} 世界の関数 f と $\text{Maybe} \llbracket \mathbf{a} \rrbracket$ 世界の関数 g は無関係ではなく

$$v? = g u? = f \sqcup u? \quad (8.6)$$

であり、

$$g = f \sqcup \quad (8.7)$$

である。つまりマップ演算子 \sqcup が関数 f を \mathbf{a} 世界から $\text{Maybe} \llbracket \mathbf{a} \rrbracket$ 世界へとジャンプさせているのである。

いま「世界」と呼んだものを、数学者は**圏**と呼ぶ。圏とは**対象**と**射**の組み合わせである。本書では「対象」とは型のことであり、射とは関数だと思えば良い。(厳密にはコンテナに入れられた関数も射に含まれる。)そして、圏から圏へとジャンプさせるものを**関手**と呼ぶ。この例で言えば値コンストラクタ $\text{Just} \llbracket x \rrbracket$ とマップ演算子 \sqcup が関手である。値コンストラクタ $\text{Just} \llbracket x \rrbracket$ は $\mathbf{a} \rightarrow \text{Maybe} \llbracket \mathbf{a} \rrbracket$ という型を持ち、マップ演算子 \sqcup は $(\mathbf{a} \rightarrow \mathbf{b}) \rightarrow (\text{Maybe} \llbracket \mathbf{a} \rrbracket \rightarrow \text{Maybe} \llbracket \mathbf{b} \rrbracket)$ という型を持つ。^{*1}

^{*1} 関手は英語でファンクター (functor) と言うが、C++ の関数オブジェクト (function object) もかつてはファンクター (functor) と呼ばれていた。C++ のファンクターとはクロージャの代用品のことで、本書で述べる関手とは異なる概念である。混同しないように注意しよう。

同じことはリストにも言える．値コンストラクタ $[x]$ とマップ演算子 \odot もまた関手である．この場合値コンストラクタは $\mathbf{a} \rightarrow [\mathbf{a}]$ という型を持ち，マップ演算子も同じく $(\mathbf{a} \rightarrow \mathbf{b}) \rightarrow ([\mathbf{a}] \rightarrow [\mathbf{b}])$ という型を持つ．

* * *

Haskell ではマップ演算子が定義された型を関手（型）と呼ぶ．具体的には，マップ演算子が定義された全ての型は **Functor** 型クラスのインスタンスであるとする．つまり，**Functor** 型クラスには一般化されたマップ演算子が定義されており，そのインスタンスであるリストや Maybe は独自のマップ演算子を定義しなければならないということである．

一般化されたマップ演算子を \cdot で表そう．この \cdot 演算子は

$$(\cdot) :: \mathbf{Functor} \sqsupset f \Rightarrow (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow f [[\mathbf{a}]] \rightarrow f [[\mathbf{b}]] \quad (8.8)$$

という型を持つ．ここに $\mathbf{Functor} \sqsupset f$ は， f が **Functor** 型クラスに属するという制約を表している．また f は型コンストラクタであり， $f [[\mathbf{a}]]$ は f 型コンストラクタと \mathbf{a} 型によって作られたコンテナ型である．

もし型コンストラクタがリスト型コンストラクタであれば，つまり $f = \mathbf{List}$ であれば

$$(\odot) :: (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow [\mathbf{a}] \rightarrow [\mathbf{b}] \quad (8.9)$$

であるし，もし型コンストラクタが Maybe 型コンストラクタであれば，つまり $f = \mathbf{Maybe}$ であれば

$$(\boxdot) :: (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow \mathbf{Maybe} [[\mathbf{a}]] \rightarrow \mathbf{Maybe} [[\mathbf{b}]] \quad (8.10)$$

である．

リストと Maybe は両者ともマップ演算子（と値コンストラクタ）を持つ．両者の関係をまとめてみたのが表 8.1 である．オブジェクト指向プログラマなら，リストと Maybe に共通のスーパークラスを設計したくなるであろう．それが型クラス **Functor** である．

表 8.1 リストと Maybe の関係

型	型コンストラクタ	マップ	値コンストラクタ
$[a]$ Maybe $\llbracket a \rrbracket$	List Maybe	\odot \square	$[x]$ $\text{Just } \llbracket x \rrbracket, \emptyset$

8.2 アプリカティブ関手

マップ演算子をさらに汎用性のあるものにするために新しく考え出された演算子がアプリカティブマップ演算子である。

いま関数のリスト $[f, g, h]$ と変数のリスト $[x, y, z]$ があるとする。リストのアプリカティブマップ演算子 \otimes を次のように定義する。

$$[f, g, h] \otimes [x, y, z] = [f x, f y, f z, g x, g y, g z, h x, h y, h z] \quad (8.11)$$

リストのアプリカティブマップ演算子はこのように、左引数のリスト内のすべての関数を順番に右引数のリスト内の変数に適用し、その結果をリストとして返す。

リストのアプリカティブマップ演算子 \otimes の型は $[a \rightarrow b] \rightarrow [a] \rightarrow [b]$ である。これは

$$\otimes :: \underbrace{a \mapsto b}_{[f_0, f \cdots f_n]} \mapsto \underbrace{a}_{[x_0, x_1 \cdots x_n]} \mapsto \underbrace{b}_{[f_0 x_0, f_0 x_1 \cdots f_0 x_n, f x_0, f x_1 \cdots f x_n]} \quad (8.12)$$

と解釈すれば良い。

リストバージョンのアプリカティブマップ演算子 (\otimes) の特別な場合として、左引数のリストの要素数が 1 の場合を考えると

$$[f] \otimes [x, y, z] = [f x, f y, f z] \quad (8.13)$$

であり、通常のマップ演算子 (\odot) を使ったマップすなわち

$$f \odot [x, y, z] = [f x, f y, f z] \quad (8.14)$$

と右辺が一致する．つまり，マップ演算子はアプリカティブマップ演算子の特別な場合と考えることができる．実際，リストマップ演算子はアプリカティブマップ演算子から

$$f \odot x_s = [f] \otimes x_s \quad (8.15)$$

と定義できる．

Maybe バージョンについても考えてみよう．Maybe に包まれた関数 $i_?$ を Maybe な変数 $u_?$ にマップするアプリカティブマップ演算子 \boxtimes を

$$i_? \boxtimes u_? = \text{case } i_? \text{ of } \begin{cases} \text{Just } [j] & \dashrightarrow j \boxtimes u_? \\ - & \dashrightarrow \emptyset \end{cases} \quad (8.16)$$

で定義する．この Maybe バージョンのアプリカティブマップ演算子 (\boxtimes) から Maybe バージョンのマップ演算子 (\boxdot) は

$$f \boxdot u_? = \text{Just } [f] \boxtimes u_? \quad (8.17)$$

のように導出できる．

これらの関係を一般化して

$$f \cdot w_* = [f] \times w_* \quad (8.18)$$

となるような一般アプリカティブマップ演算子 (\times) を考える．ここに f は関数， w_* はリストや Maybe といったコンテナ型の変数すなわち コンテナ変数 である．一般アプリカティブマップ演算子 (\times) から一般マップ演算子 (\cdot) を導き出すには，式 (8.18) のように値コンストラクタが必要である．この一般化された値コンストラクタを ピュア演算子 と呼ぶ．アプリカティブマップ演算子とピュア演算子を持つ型クラスを アプリカティブ関手 と呼び，*Applicative* 型クラスと定義する．^{*2}

ピュア演算子をピュア値コンストラクタと呼ばないのは，単純に「ピュア値」というものがないからである．*Functor* 型クラスはリスト型や Maybe

^{*2} Haskell では一般アプリカティブマップ演算子を $\langle * \rangle$ と書く．

型を抽象化したものであって、直接変数を生成できない。型クラスは、C++ の用語で言えば純粋仮想クラスのようなものであるし、Objective-C の用語で言えばメタクラスであるからである。もちろんリストのピュア演算子は $[x]$ であるし、Maybe のピュア演算子は $\text{Just}[x]$ であり、それぞれ具体的な変数を生成する。しかし変数 x にピュア演算子を適用した $[x]$ は抽象的な概念であり、そのような変数は実在しない。^{*3}

一般アプリカティブマップ演算子 (\times) は多様性によってそれぞれリストバージョンのアプリカティブマップ演算子 (\otimes) や Maybe バージョンのアプリカティブマップ演算子 (\boxtimes) にオーバーライドされ、それぞれリスト値コンストラクタ ($[x]$)、Maybe 値コンストラクタ ($\text{Just}[x]$) を用いることでリストバージョンのマップ演算子 (\odot)、Maybe バージョンのマップ演算子 (\boxdot) を生成することができる。リスト値コンストラクタ、Maybe 値コンストラクタはそれぞれピュア演算子 ($[x]$) をオーバーライドしたものであるから、結局、一般アプリカティブマップ演算子とピュア演算子のふたつがあれば、任意のクラスのマップ演算子を生成することができる。

アプリカティブマップ演算子、ピュア演算子に一般化されたバージョンがあるように、リストの $[]$ や Maybe の \emptyset を一般化した値が必要である。それを \emptyset とする。 \emptyset には特段名前が無いので、本書では単に「空」と呼ぶことにしよう。

* * *

この節の最後に[アプリカティブスタイル](#)という記法を紹介しておこう。アプリカティブマップ演算子は連続して

$$w_* = [f] \times u_* \times v_* \quad (8.19)$$

のように使える。もし $u \equiv \emptyset$ もしくは $v \equiv \emptyset$ であれば式の値は \emptyset になる。式 (8.19) からピュア演算子を消すには、最初のアプリカティブマップ演算

^{*3} Haskell は一般のピュア演算子の実装を与えていない。変数の型に応じて対応する関数が適用される。

子をマップ演算子に置き換えて

$$w_* = f \cdot u_* \times v_* \quad (8.20)$$

とすれば良い．このようにアプリカティブマップ演算子を並べる書き方をアプリカティブスタイルと呼ぶ．

8.3 関手としての関数

関数は関手である．関手とはマップ演算子を持つ型クラスのことであった．そこで、関数がどのようなマップ演算子を持つのか考えてみる．

いま関数 f が

$$f :: \mathbf{r} \rightarrow \mathbf{a} \quad (8.21)$$

という型を持っているとする．この式は \rightarrow を二項演算子，すなわち 2 引数関数とみなせば

$$f :: (\rightarrow) \mathbf{r} \mathbf{a} \quad (8.22)$$

と等価である．全く形式的に、 $((\rightarrow) \mathbf{r})$ なる型コンストラクタがあるとして

$$\mathbf{r} \rightarrow \mathbf{a} = ((\rightarrow) \mathbf{r}) \mathbf{a} \quad (8.23)$$

であると考えてみる．型 \mathbf{a} から型コンストラクタ $((\rightarrow) \mathbf{r})$ によって型 $(\mathbf{r} \rightarrow \mathbf{a})$ が作られると考えるのだ．^{*4}

マップ演算子の型は、 $((\rightarrow) \mathbf{r}) \mathbf{a} = ((\rightarrow) \mathbf{r}) \llbracket \mathbf{a} \rrbracket$ とすると

$$(\cdot) :: \text{Functor } \mathbf{f} \Rightarrow (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow ((\rightarrow) \mathbf{r}) \llbracket \mathbf{a} \rrbracket \rightarrow ((\rightarrow) \mathbf{r}) \llbracket \mathbf{b} \rrbracket \quad (8.24)$$

であって、関数のマップ演算子を \circ とすると

$$(\circ) :: (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow ((\rightarrow) \mathbf{r}) \llbracket \mathbf{a} \rrbracket \rightarrow ((\rightarrow) \mathbf{r}) \llbracket \mathbf{b} \rrbracket \quad (8.25)$$

^{*4} Haskell では $((\rightarrow) \mathbf{r})$ を $((\rightarrow) \mathbf{r})$ と書く．

であり, これはすなわち

$$(\circ) :: (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow (\mathbf{r} \rightarrow \mathbf{a}) \rightarrow (\mathbf{r} \rightarrow \mathbf{b}) \quad (8.26)$$

のことである.

いま関数 $f :: \mathbf{r} \rightarrow \mathbf{a}$ とは別な関数 $g :: \mathbf{a} \rightarrow \mathbf{b}$ があったとしよう. 関数 f と関数 g の合成 $g \cdot f$ の型は

$$g \cdot f :: \mathbf{r} \rightarrow \mathbf{b} \quad (8.27)$$

であるから,

$$(\cdot) :: (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow (\mathbf{r} \rightarrow \mathbf{a}) \rightarrow (\mathbf{r} \rightarrow \mathbf{b}) \quad (8.28)$$

である. つまり関数のマップ演算子 (\circ) と関数の合成演算子 (\cdot) は同じ型を持つ.

幸い, 我々は関数のマップ演算子の実装に関しては, 型さえ守っていれば (そして第 9 章で述べる関手則さえ守っていれば) 自由に選べる. そこで

$$g \circ f = g \cdot f \quad (8.29)$$

としておこう. これは

$$g \circ f = \lambda x \mapsto g(fx) \quad (8.30)$$

と書いても同じことである. これが Haskell における関数のマップ演算子の定義である.

* * *

関数はアプリカティブ関手でもある. アプリカティブ関手には, アプリカティブマップ演算子とピュア演算子が定義されるのであった. そこで, 関数版のアプリカティブマップ演算子を \bowtie とし, 関数版のピュア演算子を $\langle x \rangle$ と書くことにしよう.

ピュア演算子は $\mathbf{a} \rightarrow (\mathbf{r} \rightarrow \mathbf{a})$ 型を持たなければならない. 従って関数版のピュア演算子は変数から関数を作るとも考えられる. 我々は関数版のピュア演算子として

$$\langle x \rangle = \backslash _ \mapsto x \quad (8.31)$$

を採用する.*5

関数版のアプリカティブマップ演算子を \bowtie とすると, その型は

$$(\bowtie) :: ((\rightarrow)^{\mathbf{r}}) [\mathbf{a} \rightarrow \mathbf{b}] \rightarrow ((\rightarrow)^{\mathbf{r}}) [\mathbf{a}] \rightarrow ((\rightarrow)^{\mathbf{r}}) [\mathbf{b}] \quad (8.32)$$

つまり

$$(\bowtie) :: (\mathbf{r} \rightarrow \mathbf{a} \rightarrow \mathbf{b}) \rightarrow (\mathbf{r} \rightarrow \mathbf{a}) \rightarrow (\mathbf{r} \rightarrow \mathbf{b}) \quad (8.33)$$

である.

我々は関数版アプリカティブマップ演算子として

$$g \bowtie f = \backslash x \mapsto gx(\textcolor{teal}{f}x) \quad (8.34)$$

とする. これは, 関数版のピュア演算子の定義と, 一般マップ演算子と一般アプリカティブマップ演算子の関係 $f \cdot w = \llbracket f \rrbracket \times w$ から導かれる. すなわち

$$\langle g \rangle \bowtie f = \backslash x \mapsto \langle g \rangle x(\textcolor{teal}{f}x) \quad (8.35)$$

$$= \backslash x \mapsto (\backslash _ \mapsto g)x(\textcolor{teal}{f}x) \quad (8.36)$$

$$= \backslash x \mapsto g(\textcolor{teal}{f}x) \quad (8.37)$$

$$= g \cdot f \quad (8.38)$$

であるからである.

*5 Haskell では $\langle x \rangle$ を `const x` と書く.

8.4 余談：アプリカティブマップ演算子の実装

リストと Maybe のアプリカティブマップ演算子は、それぞれのマップ演算子から定義することができる．リストのアプリカティブマップ演算子の定義は次の通り．

$$\begin{cases} [] \otimes x_s = [] \\ (f : f_s) \otimes x_s = \flat((f \odot x_s) : (f_s \otimes x_s)) \end{cases} \quad (8.39)$$

ここに $(f : f_s)$ は関数のリストであり、 x_s はリスト変数である．

Maybe のアプリカティブマップ演算子の定義は次の通り．

$$g? \boxtimes u? = \text{case } g? \text{ of } \begin{cases} \text{Just } \llbracket h \rrbracket & \dashrightarrow h \sqcap u? \\ - & \dashrightarrow \emptyset \end{cases} \quad (8.40)$$

ここに $g?$ は Maybe コンテナに入れられた関数、 $u?$ は Maybe 変数である．

8.5 この章のまとめ*

1. ...

第 9 章

モナド*

9.1 バインド演算子

一般マップ演算子をピュア演算子と一般アプリカティブマップ演算子に分解することで、式の見通しを良くすることができるアプリカティブスタイルという記法を採用できた。アプリカティブスタイルでは

$$f \cdot u_* \times v_* \times w_* \tag{9.1}$$

という風にコンテナ変数 u_*, v_*, w_* に関数 f を適用させることができる。コンテナ変数 u_*, v_*, w_* のいずれかが \emptyset であれば式全体の値が \emptyset になる。これは 3 個の計算を並列に行って、その結果をそれぞれ u_*, v_*, w_* に入れておき、最後に関数 f に投げるという計算構造を具現化したものである。(関数 f は C で言えば main 関数に相当するであろう。)

しかしながら、アプリカティブスタイルでは変数に文脈を与えるタイミングがコンテナ変数を作るときのそれぞれ 1 回に限られている。そこで、任意のタイミングで変数に文脈を与えられるように、別な方法で一般マップ演算子を分解してみよう。

Maybe の例を思い出そう。Maybe 型の変数 $u_?$ はラップされた値 $\text{Just}[x]$ を持つのか、エラーを表す \emptyset を持つのかを選べる。そこで、引数 x をとり

何らかの計算をする関数 g を考えよう．この関数 g は引数 x の値次第ではエラーを表す \emptyset を返す．例えば

$$\begin{aligned} gx \mid_{x \neq 0} &= \text{Just} \llbracket 1/x \rrbracket \\ &\mid_{\text{otherwise}} = \emptyset \end{aligned} \quad (9.2)$$

といった関数が考えられる．変数 x は文脈を持っていないが，関数 g を適用した結果である gx は文脈を持っていることに注意しよう．いま gx は Maybe という文脈を持っているから，我々は

$$v? = gx \quad (9.3)$$

という風に結果を Maybe 変数に保存しなければならない．今まで見てきた $y = fx$ や $v? = f \sqcup u?$ の関係とは異なることに注意しよう．

関数 g の型は

$$g :: \mathbf{a} \rightarrow \text{Maybe} \llbracket \mathbf{a} \rrbracket \quad (9.4)$$

である．ということは，関数 g を Maybe 変数に適用させようと思っても，我々が既に知っているマップ演算子 \sqcup やアプリアティブマップ演算子 \boxtimes が使えないということである．前者は第1引数に $\mathbf{a} \rightarrow \mathbf{b}$ 型の関数を取るし，後者は第1引数に $\text{Just} \llbracket (\mathbf{a} \rightarrow \mathbf{b}) \rrbracket$ 型の関数 (Maybe 関数) を取るからである．

そこで，新しいマップ演算子を発明する．いま Maybe 変数 $u?$ を $u? = \text{Just} \llbracket x \rrbracket$ としよう．新しいマップ演算子 \spadesuit を使って

$$v? = g \spadesuit u? \quad (9.5)$$

とする．この新しいマップ演算子 \spadesuit のことを Maybe の バインド演算子 と呼ぶ．ここで，もし計算が成功していたら $v? = \text{Just} \llbracket gx \rrbracket$ であり，失敗していたら $v? = \emptyset$ である．

演算 $g \spadesuit u?$ の結果は Maybe 値であるから，バインド演算子は連続して用いることができる．通常の引数を取って Maybe 値を返すもう一つの関数 h

があるとする

$$v_? = h \spadesuit (g \spadesuit u_?) \quad (9.6)$$

のように連続して関数を適用できる。バインド演算子は右結合するので、上式は

$$v_? = h \spadesuit g \spadesuit u_? \quad (9.7)$$

のように簡潔に書ける。このスタイルなら演算子もすべて統一できていて、かつどの関数でも戻り値を \emptyset に切り替えられるので、アプリカティブスタイルよりも強力と言える。

具体例で考えてみよう。関数 g を

$$\begin{aligned} gx \mid_{x \neq 0} &= \text{Just} \llbracket 1/x \rrbracket \\ \mid_{\text{otherwise}} &= \emptyset \end{aligned} \quad (9.8)$$

とする。また、関数 h を

$$\begin{aligned} hy \mid_{-\frac{\pi}{2} < y < \frac{\pi}{2}} &= \text{Just} \llbracket \tan y \rrbracket \\ \mid_{\text{otherwise}} &= \emptyset \end{aligned} \quad (9.9)$$

とする。このとき $u_? = \text{Just} \llbracket 4/\pi \rrbracket$ とすると

$$v_? = h \spadesuit g \spadesuit u_? \quad (9.10)$$

の計算結果として $v_? = \text{Just} \llbracket 1 \rrbracket$ を得る。一方で $u_? = \text{Just} \llbracket 0 \rrbracket, u_? = \text{Just} \llbracket 1/\pi \rrbracket, u = \emptyset$ などの場合は $v_? = \emptyset$ となり、計算できなかったという結果を得る。

というわけで、**Applicative** 型クラスをさらに拡張して、一般のバインド演算子を持たせることを考えてみよう。我々はこの新しい型クラスを モナド 型クラスと呼び **Monad** で表す。

9.2 モナド

Functor 型クラスは一般マップ演算子 (\cdot) を持っていた。 **Applicative** 型クラスはピュア演算子 $(\llbracket x \rrbracket)$ と一般アプリカティブマップ演算子 (\times) を持っており、このふたつの演算子から一般マップ演算子を合成できた。新しい **Monad** 型クラスは、一般バインド演算子 (\heartsuit) とピュア演算子を持つものとしよう。後で見るように、一般アプリカティブマップ演算子はピュア演算子と一般バインド演算子から合成できる。^{*1}

関数 i が次の形をしているとする。

$$ix = \text{case } x \text{ of } \begin{cases} \text{lovely} & \dashrightarrow \llbracket fx \rrbracket \\ - & \dashrightarrow \emptyset \end{cases} \quad (9.11)$$

ここに x は非コンテナ変数で、関数 f も「普通の」(コンテナに入っていない) 関数である。より厳密に言えば $x :: \mathbf{a}$ かつ $f :: \mathbf{a} \rightarrow \mathbf{b}$ である。従って、関数 i_* の型は $\mathbf{a} \rightarrow \llbracket \mathbf{b} \rrbracket$ である。条件 **lovely** には任意の値を入れてよい。

関数 i をコンテナに入った変数 $u = \llbracket x \rrbracket$ に適用させるのが一般バインド演算子 (\heartsuit) の役割である。計算結果をコンテナ変数 a に格納するとすると、関数 i のコンテナ変数 u_* への適用は

$$a_* = i \heartsuit u_* \quad (9.12)$$

と書ける。うまく行けば $a_* = \llbracket fx \rrbracket$ となるし、そうでなければ $a_* = \emptyset$ となる。

^{*1} Haskell には最初に関手が導入され、その次に関手を拡張する形でモナドが導入された。そしてその次に、関手を拡張しなおす形でアプリカティブ関手が導入された。それゆえ、モナドとアプリカティブ関手には概念的重複があるにもかかわらず、別々に定義されるという悲劇が暫くの間続いた。アプリカティブ関手のピュア演算子と、モナドのピュア演算子は概念的に同じものであるにもかかわらず、別々の演算子として定義されていたのである。この状態は GHC v7.10 以降で、モナドがアプリカティブ関手を拡張する形に改められたことで解消した。ただし、かつて「モナド版のピュア演算子」としてモナドに「ユニット演算子」が定義されていたことから、ピュア演算子のことをユニット演算子と呼ぶ場合がある。

さて $u_* = \llbracket x \rrbracket, v_* = \llbracket y \rrbracket$ として, かつ $y = fx$ であるとき

$$v_* = f \cdot u_* = \llbracket f \rrbracket \times u_* \quad (9.13)$$

であった.

~~TK::CHECK:~~

式 (9.11) から $ix = \llbracket fx \rrbracket$ すなわち $i = \llbracket f \diamond \rrbracket$ の関係を抜き出すと式 (9.12) は

$$a_* = \llbracket f \diamond \rrbracket \heartsuit u_* \quad (9.14)$$

となる. いま $y = fx$ であったから $a_* = v_*$ であり, 最終的に

$$v_* = f \cdot u_* = \llbracket f \rrbracket \times u_* = \llbracket f \diamond \rrbracket \heartsuit u_* \quad (9.15)$$

を得る. これが一般バインド演算子と一般アプリカティブマップ演算子, 一般マップ演算子の関係である.

* * *

~~TK::To be written:~~

リストモナド

$$\llbracket x \rrbracket = [x] \quad (9.16)$$

$$f :: \mathbf{a} \rightarrow [\mathbf{a}] \quad (9.17)$$

$$f \heartsuit x_s = \mathbf{b}(f \odot x_s) \quad (9.18)$$

Maybe モナド

$$\llbracket x \rrbracket = \text{Just } [x] \quad (9.19)$$

$$f :: \mathbf{a} \rightarrow \text{Maybe} \llbracket \mathbf{a} \rrbracket \quad (9.20)$$

$$f \heartsuit u? = \text{case } u? \text{ of } \begin{cases} \text{Just } \llbracket x \rrbracket & \dashrightarrow \text{Just } \llbracket f x \rrbracket \\ - & \dashrightarrow \emptyset \end{cases} \quad (9.21)$$

* * *

~~TK:~~ To be written:

$$f \spadesuit g = f \heartsuit (g \diamond) \quad (9.22)$$

9.3 関手則・アプリカティブ関手則・モナド則

関手，アプリカティブ関手，モナドにはそれぞれ従う規則がある．これらは自然法則ではなく，定義である．

関手の一般マップ演算子 (\cdot) は次の規則に従う．

1. 単位元の存在: $\text{id} \cdot u = u$
2. 合成則: $(f \cdot g) \cdot = (f \cdot) \cdot (g \cdot)$

ただし関数 id は $\text{id } x = x$ で定義される．もちろん $\text{id} = \diamond$ と定義しても同じである．

例えば $\text{id} \cdot x_s$ は $\text{id } x_s$ であり，結局は x_s である．単位元の存在とは，そのような関数 id があるという規則である．

合成則のほうは $x_s = [x]$ を例に考えるとわかりやすく,

$$((f \cdot) \cdot (g \cdot))x_s = (f \cdot (g \cdot \diamond))x_s \quad (9.23)$$

$$= f \cdot (g \cdot x_s) \quad (9.24)$$

$$= f \cdot [gx] \quad (9.25)$$

$$= [(f \cdot g)x] \quad (9.26)$$

$$= (f \cdot g) \cdot [x] \quad (9.27)$$

$$= (f \cdot g) \cdot x_s \quad (9.28)$$

のような関係を一般化したものだと考えれば良い.

合成則は Haskell コンパイラによって最適化のために積極的に使われる.

* * *

アプリカティブ関手の一般アプリカティブマップ演算子 (\times) およびピュア演算子は次の規則に従う.

1. 単位元の存在: $[\text{id}] \times u = u$
2. 準同型則: $[f] \times [x] = [fx]$
3. 合成則: $[(\cdot)] \times \phi \times \psi \times v = \phi \times (\psi \times v)$
4. 交換則: $\varphi \times [y] = [(\$y)] \times \varphi$

ここでも $\text{id} = \diamond$ である.

アプリカティブマップ演算子の準同型則は $w = [fx]$ とした時に,

$$\begin{array}{ccc} x & \xrightarrow{[\cdot]} & [x] \\ f \downarrow & & \downarrow [f] \times \\ fx & \xrightarrow{[\cdot]} & w \end{array} \quad (9.29)$$

のように x からスタートして, どちらのルートを辿っても w に行き着くという意味である.

アプリカティブマップ演算子の合成則も注釈が必要であろう. 仮に $\phi =$

$\llbracket g \rrbracket, \psi = \llbracket h \rrbracket, v = \llbracket z \rrbracket$ とすると、合成則の左辺は

$$\llbracket (\cdot) \rrbracket \times \phi \times \psi \times v = \llbracket (\cdot)g \rrbracket \times \psi \times v \quad (9.30)$$

$$= \llbracket (\cdot)gh \rrbracket \times v \quad (9.31)$$

$$= \llbracket g \cdot h \rrbracket \times v \quad (9.32)$$

$$= \llbracket g \cdot hz \rrbracket \quad (9.33)$$

となる一方、合成則の右辺は

$$\phi \times (\psi \times v) = \phi \times \llbracket hz \rrbracket \quad (9.34)$$

$$= \llbracket g \cdot hz \rrbracket \quad (9.35)$$

となり一致する．合成則とは、このような関係が満たされるように一般アプリケーションマップ演算子を定義しておきなさいという意味だ．

* * *

モナドの一般バインド演算子 (\heartsuit) は次の規則に従う．

1. 右単位元の存在: $i \heartsuit \llbracket x \rrbracket = ix$
2. 左単位元の存在: $\llbracket \text{id} \rrbracket \heartsuit u = u$
3. 結合則: $i \heartsuit (j \heartsuit v) = (i \heartsuit (j \diamond)) \heartsuit v$

結合則についてのみ解説しておこう．

$$i \heartsuit \llbracket z \rrbracket = \llbracket fz \rrbracket, j \heartsuit \llbracket z \rrbracket = \llbracket gz \rrbracket, v = \llbracket y \rrbracket \quad (9.36)$$

とすると

$$i \heartsuit (j \heartsuit v) = i \heartsuit \llbracket gy \rrbracket \quad (9.37)$$

$$= \llbracket f(gy) \rrbracket \quad (9.38)$$

$$= \llbracket f \cdot gy \rrbracket \quad (9.39)$$

である一方, $k = i \heartsuit (j \diamond)$ とすると

$$kz = (i \heartsuit (j \diamond))z \quad (9.40)$$

$$= i \heartsuit (jz) \quad (9.41)$$

$$= \llbracket f(gz) \rrbracket \quad (9.42)$$

$$= \llbracket f \cdot gz \rrbracket \quad (9.43)$$

であるから

$$k = i \heartsuit (j \diamond) = \llbracket f \cdot g \diamond \rrbracket \quad (9.44)$$

を得る. ここで

$$\llbracket f \cdot g \diamond \rrbracket \heartsuit v = (f \cdot g) \cdot v \quad (9.45)$$

$$= \llbracket f \cdot gy \rrbracket \quad (9.46)$$

であるから, 結合則

$$i \heartsuit (j \heartsuit v) = (i \heartsuit (j \diamond)) \heartsuit v \quad (9.47)$$

$$= (i \boxtimes j) \heartsuit v \quad (9.48)$$

を得ることになる.

9.4 余談：モナドとしての関数

関数はアプリカティブ関手であった. 関数のピュア演算子とアプリカティブマップ演算子はそれぞれ

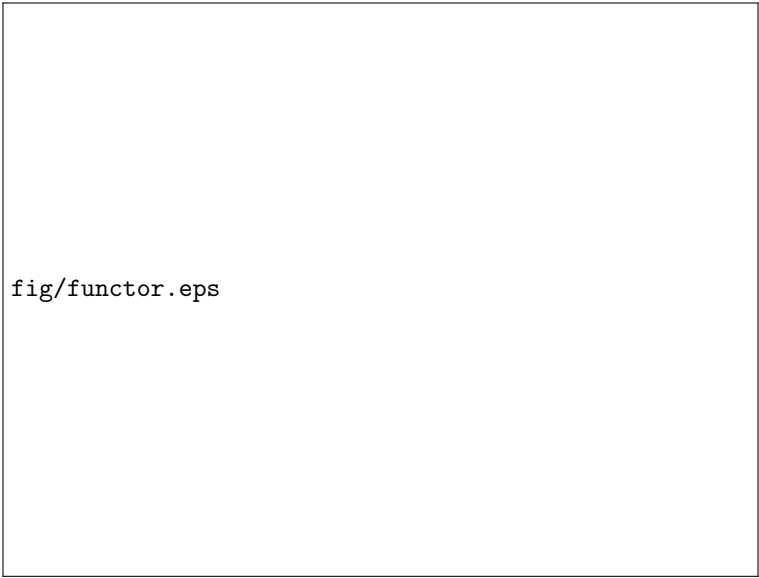
$$\langle x \rangle = \backslash _ \mapsto x \quad (9.49)$$

$$g \bowtie f = \backslash x \mapsto g \textcolor{blue}{x}(\textcolor{red}{f} x) \quad (9.50)$$

であった.

関数のバインド演算子 \diamond を考えておこう. 関数のバインド演算子は

$$g \diamond f = \backslash x \mapsto g(\textcolor{red}{f} x)x \quad (9.51)$$



fig/functor.eps

図 9.1 ...

と定義する。ピュア演算子とバインド演算子から、関数のマップ演算子を

$$\langle g \diamond \rangle \diamond f = \lambda x \mapsto \langle g \diamond \rangle (f x) x \quad (9.52)$$

$$= \lambda x \mapsto \langle g(f x) \rangle x \quad (9.53)$$

$$= \lambda x \mapsto (\lambda _ \mapsto g(f x)) x \quad (9.54)$$

$$= \lambda x \mapsto g(f x) \quad (9.55)$$

$$= g \cdot f \quad (9.56)$$

$$= g \circ f \quad (9.57)$$

のように合成できる。

表 9.1 型と型クラスの関係

型 \ 型クラス	MonadPlus					
	Monad				Monoid	
	Applicative					
	Functor					
一般モノイド					\emptyset	+
整数					0	+
整数					1	*
一般コンテナ	♡	$\llbracket x \rrbracket$	\times	\cdot		
リスト	♣	$[x]$	\otimes	\odot	$[]$	\oplus
Maybe	♠	$\text{Just } \llbracket x \rrbracket$	\boxtimes	\boxdot	\emptyset	(x の型に依存)
関数	◇	$\langle x \rangle$	\boxtimes	\circ	◇	\cdot

まとめると

$$\langle x \rangle = \backslash _ \mapsto x \quad (9.58)$$

$$g \circ f = \backslash x \mapsto g(\textcolor{teal}{f} x) = g \cdot f \quad (9.59)$$

$$g \boxtimes f = \backslash x \mapsto g x (\textcolor{teal}{f} x) \quad (9.60)$$

$$g \diamond f = \backslash x \mapsto g(\textcolor{teal}{f} x) x \quad (9.61)$$

である.

* * *

関数はモノイドとしての性質も持つ. 関数 $\text{id} = \diamond$ とすると, 任意の関数 f に対して

$$\text{id} \cdot f = f \cdot \text{id} = f \quad (9.62)$$

であるから, 関数全体の集合を \mathbb{F} で表すと, 組み合わせ $(\mathbb{F}, \cdot, \text{id})$ はモノイドである.

モナドでありモノイドである型クラスをモナドプラスと呼ぶ。関数はモナドプラスである。

今まで出てきた型と型クラスの関係を表 9.3 に示す。Maybe に関しては **a** 型がモノイドである場合に限って $\text{Maybe}[\text{a}]$ 型もモノイドである。

9.5 この章のまとめ*

1. ...

第 10 章

IO*

A...

10.1 アクション

計算機の状態を変えることを**副作用**と呼ぶ。副作用とは変数への破壊的代入に他ならない。例えば、計算機は画面やプリンタに何かを出力するが、それは画面やプリンタという「変数」を書き換えていることになる。また例えば擬似乱数の生成も副作用である。呼び出されるたびに異なる値を返す擬似乱数生成関数は、そのたびに計算機の内部状態を書き換えているのである。

副作用を持つ関数を**アクション**と呼ぶ。アクションは値を持つが、その値は計算の実行時までわからない。例えば擬似乱数を生成するアクション `rand` があるとしよう。これを変数 α に

$$\alpha = \text{rand} \tag{10.1}$$

と代入しても、変数 α に擬似乱数が代入されるわけではない。「擬似乱数を生成する」というアクションが α に代入されたのだ。

では、いつ「擬似乱数を生成する」アクションが実行されるのだろうか。それは、プログラムがまさに計算機の状態を変えるタイミング、つまりプロ

グラムが実行されるタイミングなのである．そのためには，プログラムそのものをアクションで表しておかないといけない．我々はプログラム全体を ω で表すことにしよう．

プログラムが計算機状態を変える一例として，画面に値を出力することを考える．画面に値を出力するアクションを `putStr` と名付けよう．例えば次のプログラム例は，画面に “Hello, world.” と書き出すものとする．

$$\omega = \text{putStr } s \text{ where } s \triangleq \text{“Hello, world.”} \quad (10.2)$$

アクション `putStr` は変数をつとり，その値を画面へ出力すなち破壊的代入を行う．では $\omega = \text{putStr } \alpha$ とすればアクション α が実行されて，晴れて擬似乱数が生成され，その値が画面へ出力されるだろうか．もちろんそうはならないのである．

ここに α はアクションであり，変数ではない．一方で，アクション `putStr` は変数を受け取る．つまり，アクションから何らかの方法で値を「安全に」抜き取らないといけない．ここで安全性にこだわるのは，アクション α が副作用を持つからである．副作用を参照透過な変数へ伝播させてはいけない．副作用を持つアクションは，副作用を持つアクションへのみ受け継がれなければならない．

この話は何かと似ていないだろうか．そう，Maybe である．一度ゼロ除算の可能性に汚染されたコンテナ変数は，コンテナから出すことが許されないのである．Maybe を返す関数 f の戻り値

$$\begin{aligned} fx|_{x \neq 0} &= \text{Just } [1/x] \\ |_{\text{otherwise}} &= \emptyset \end{aligned} \quad (10.3)$$

を，別の関数

$$gy = 1 + y \quad (10.4)$$

に渡そうと思ったら，

$$g \heartsuit fx \quad (10.5)$$

のようにバインド演算子で合成しなければならなかった。

我々はアクションにもバインド演算子を拡張して、

$$\omega = \text{putStr} \heartsuit \alpha \quad (10.6)$$

とする。これは第9章で見たバインド演算子と同じものである。同じバインド演算子が使えるカラクリは、次節で見ていくことにする。

10.2 IO モナド

ある関数 c が引数を取らず、いつも決まった **Double** 型の数を返すとしてよう。そうすると、関数 c の型は単純に **Double** である。

呼び出すたびに異なる擬似乱数を返すアクション **rand** もまた毎回 **Double** 型の数を返す。そこで **rand** も **Double** 型としたいところだが、こちらは関数ではなくアクションである。そこを区別するために、**rand** の型は $\text{IO}[\text{Double}]$ 型と $\text{IO}[\dots]$ に入れて区別する。^{*1}

呼び出しても何も返さないアクションはどのような型を持つべきだろう。何も返さない関数というものは無意味だが、アクションは副作用を持つので、何も返さないものがあっても良いのである。何も返さないことを「空っぽ」を返すと読み替えて、何も返さないアクションの型を $\text{IO}[(\)]$ 型としよう。ここに $(\)$ は「空っぽ」の意味で、ユニット型と読む。 $\text{IO}[(\)]$ 型のアクションの例は、画面に値を出力する **putStr** アクションである。^{*2}

キーボードからの入力を受け取るアクションもある。そのアクションを **readLn** としよう。アクション **readLn** は文字列型を返すので、その型は $\text{IO}[\text{String}]$ である。^{*3}

アクション **putStr**, **readLn**, **rand** は計算機の状態を変化させる。アクション **putStr**, **readLn** は OS のシステムコールを発行して、前者ならビデオメ

^{*1} Haskell では $\text{IO}[\text{Double}]$ のことを **IO Double** と書く。

^{*2} Haskell では $\text{IO}[(\)]$ のことを **IO ()** と書く。

^{*3} Haskell では $\text{IO}[\text{String}]$ のことを **IO String** と書く。

モリの値を書き換えるし、後者ならシリアルインタフェースの入力バッファをフラッシュする。アクション `rand` は呼ばれるたびに内部のカウンタ値を一つ進める。これらのアクションは参照透過性を破壊する。そのために、プログラムの他の部分から隔離されねばならない。その隔離のメカニズムを提供するのがモナドである。

[a] 型が **a** 型のリストであるように、 $IO[a]$ 型は **a** 型の **IO** である。そしてリストがモナドであるように、IO もまたモナドである。モナドには、バインド演算子と、アプリカティブ関手から引き継いだピュア演算子の二つが必要であった。アプリカティブ関手から引き継いだアプリカティブマップ演算子はバインド演算子とピュア演算子の二つから合成できるし、関手から引き継いだマップ演算子もまたアプリカティブマップ演算子とピュア演算子から合成できるから、アプリカティブマップ演算子とマップ演算子は改めて実装しておく必要はない。

$IO[()]$ 型の場合、ピュア演算子は何も返す必要がないから

$$[[]] = () \quad (10.7)$$

であるとする。

10.3 do 記法

モナドを使った書き方が従来のプログラムの記法からあまりにもかけ離れていることに、Haskell の設計者は気づいていたようで、Haskell には次に述べる **do 記法** という記法が用意されている。この do 記法はもちろんシンタックスシュガーで、新しいことは何もない。

例えば $\omega = \text{putStr} \heartsuit \alpha$ where $\alpha \triangleq \text{rand}$ を do 記法を用いて書き直すと

$$\omega = \text{do } \{a \leftarrow \text{rand}; \text{putStr } a\} \quad (10.8)$$

となる。ここで変数 a は $\{\dots\}$ の中でだけ参照できる変数である。バインド演算子が副作用やゼロ除算汚染を中に閉じ込めたように、do 記法の括弧

は副作用や汚染を閉じ込める役割を果たす。^{*4}

ここで、同様なことをする Python プログラムを見てみよう。

```
import random
def main:
    a = random.random() # (1)
    print(a)             # (2)
```

Python

コード中の (1) の行で $a \leftarrow \text{rand}$ を実行し、(2) の行で `putStr a` を実行していると思えば、このコードと式 (10.8) の順序はそっくり同じである。もちろんこの do 記法はバインド演算子を使った式を切り貼りして、順序を入れ替えただけである。

do 記法には \leftarrow の他に、我々の let とよく似た `let` という構文が用意されている。この `let` は局所変数の導入に用いられて、例えば

$$\omega = \text{do} \{ \text{let } y = fx; \alpha y \} \quad (10.9)$$

のように使う。

以下に、do 記法を使った例を示す。

$$\text{do} \{ \alpha x \} = \alpha x \quad (10.10)$$

$$\text{do} \{ y \leftarrow \alpha x; \beta y \} = \beta \heartsuit \alpha x \quad (10.11)$$

$$\text{do} \{ \alpha x; \beta y \} = (\backslash _ \mapsto \beta y) \heartsuit \alpha x \quad (10.12)$$

$$\text{do} \{ \text{let } y = fx; \alpha y \} = \alpha y \text{ where } y \triangleq fx \quad (10.13)$$

$$\text{do} \{ y \leftarrow \alpha x; \text{let } z = fy; \beta z \} = \beta \heartsuit (f \cdot \alpha x) \quad (10.14)$$

^{*4} Haskell では `omega = do { a <- rand; putStr a }` と書くか、または ; を改行に置き換えて

```
omega = do
    a <- rand
    putStr a
```

と書く。

より複雑な例も挙げる.

$$\begin{aligned} & \text{do } \{y \leftarrow \alpha x; y' \leftarrow \alpha' x'; \text{let } z = fyy'; \beta z\} \\ &= \beta \heartsuit (f \cdot \alpha x \times \alpha' x') \quad (10.15) \end{aligned}$$

最後に do 記法中に変数を 2 回以上使いまわす例を示す.

$$\begin{aligned} & \text{do } \{y \leftarrow \alpha x; y' \leftarrow \alpha' x'; \text{let } z = fyy'; \beta z; \text{let } z' = f'yy'; \beta' z'\} \\ &= (\backslash yy' \mapsto ((\backslash _ \mapsto \text{let } z \triangleq fyy' \text{ in } \beta z) \\ & \quad \heartsuit (\text{let } z' \triangleq f'yy' \text{ in } \beta' z')))(\alpha x)(\alpha' x') \quad (10.16) \end{aligned}$$

この例では変数 y, y' が 2 回使われている.

我々はバインド演算子を使った通常の記法と do 記法のいずれか読みやすい方を採用すれば良い.

* * *

バインド演算子には、左右の引数を入れ替えた右バインド演算子がある.
右バインド演算子は $\heartsuit\rightarrow$ と書き,

$$\alpha x \heartsuit\rightarrow \beta = \beta \heartsuit \alpha x \quad (10.17)$$

であるとする.*5

10.4 余談：関数であるということ*

IO モナドの変数は、たとえ値を読み出すだけであっても必ず関数適用が必要である. それは、IO モナドの変数が「自分が読み出されたこと」を知る必要があるからである. IO モナドの変数は、自分が読み出されたタイミングで副作用を発生させる. これは Objective-C や Swift に見られる getter メソッドと同じ考え方である.

*5 Haskell では $\alpha x \heartsuit\rightarrow \beta$ を $\alpha x \gg= \beta$ と書く.

IO モナドの変数値を読み出すために行われる関数呼び出しはダミーである場合があり、戻り値はしばしば捨てられる。

変数をダミーの関数で包み、それをさらに IO モナドで包んだのが IO モナド変数である。

もうひとつ、アクションが関数でなければならない理由がある。Haskell はいつも遅延評価を行うことを思い出してもらいたい。Python であれば、式は書かれた順に評価される。しかし、例えば

$$y_1 = f x_1 \quad (10.18)$$

$$y_2 = g x_2 \quad (10.19)$$

という式があった場合、関数 f と g のどちらが先に評価されるか、あるいは同時に評価されるかは Haskell では未定義である。Haskell で唯一計算順序が保証されているのは、関数適用である。例えば

$$y = g_2(g_1 x) \quad (10.20)$$

であれば、確実に関数 g_1 が関数 g_2 よりも先に評価される。

直列に評価したい関数の戻り値が、いつも次の関数の引数の型と一致しているとは限らないし、次の関数（アクション）が引数を取らない可能性もある。もし関数 g_2 が引数を取らなければ

$$\langle g_2 \rangle (g_1 x) \quad (10.21)$$

とする。この時、関数適用 $(g_1 x)$ の結果は単純に捨てられる。^{*6}

* * *

ふたつのアクション α_1, α_2 があり、アクション α_2 が引数を取らない場合、その二つを合成するには

$$\alpha_1 x \rightsquigarrow \langle \alpha_2 \rangle \quad (10.22)$$

^{*6} C では void 型を返す関数 `void g1(int)` を引数を取らない関数 `int g2(void)` に「食わせる」ことが可能で、`g2(g1(x))` は正しいコードである。もっとも C プログラマーは `g1(x)`, `g2()` という書き方の方を好むであろう。

とする．式 (10.22) はしばしば

$$\alpha_1 x \twoheadrightarrow \alpha_2 = \alpha_1 x \multimap \langle \alpha_2 \rangle \quad (10.23)$$

なる演算子 \twoheadrightarrow を用いて記述される．^{*7}

10.5 この章のまとめ*

1. do 記法.

^{*7} Haskell では $\alpha_1 x \twoheadrightarrow \alpha_2$ を `alpha1 x >> alpha2` と書く．

第 11 章

データ型の定義*

11.1 データ型

我々はしばしば新しい集合を考える必要に迫られる．例えば，イチ，ニ，サン，タクサンからなる集合

$$\mathbf{Num} \stackrel{\text{def}}{=} \{\text{One}, \text{Two}, \text{Three}, \text{Many}\} \quad (11.1)$$

を考えることがあるだろう．集合 \mathbf{Num} の元 $n :: \mathbf{Num}$ は One , Two , Three , Many のいずれかの値を取ることになる．

数学者は新しい集合を定義するが，Haskell プログラマは新しいデータ型を定義する．集合 \mathbf{Num} の定義の代わりに，我々はデータ型 \mathbf{Num} を

$$\text{datatype } \mathbf{Num} = \text{One} \vee \text{Two} \vee \text{Three} \vee \text{Many} \quad (11.2)$$

のように書いて定義するものとする．式の先頭にある datatype は，この式がデータ型の定義であることを示すタグである．部分的にアンダーラインが引かれているのは，Haskell が datatype を data と省略してしまうからである．これは不本意なことだが，文字数節約のために仕方ない．ここに \mathbf{Num} は型コンストラクタ， One , Two , Three , Many は値コンストラクタであ

る.*1

この節で紹介したデータ型の定義は C で言う `enum` に近い。他に C で言う `struct` や `union` すなわち構造体や共用体もこの `datatype` 文で定義できるが、これは次節で見ることにする。

* * *

新しいデータ型がある型クラスに属するとき、`datatype` 文でそれを同時に宣言できる。例えば `Num` 型は同値演算子 (`≡`) を持つことが自然である。型 `Num` が `Eq` 型クラスに属するとき、

`datatype Num = One ∨ Two ∨ Three ∨ Many deriving Eq` (11.3)

のように書く。*2

型 `Num` の同値演算子 (`≡`) の実装は 11.3 節で見ることにする。

11.2 レコード構文

データ型の定義では、値コンストラクタにパラメタを与えることもできる。例えば `Rectangle` という型を考えよう。この型は、始点の平面座標と幅、高さで合計 4 個の `Float` のパラメタを取るものとする。このとき

`datatype Rectangle`
`= Rectangle [Float Float Float Float] deriving Eq` (11.4)

という風にデータ型の定義を行う。*3

`Rencatngle` 型の変数は次のように初期化する。

$$\left\| \begin{array}{l} r :: \text{Rectangle} \\ r = \text{Rectangle} [1\ 2\ 3\ 4] \end{array} \right\| \quad (11.5)$$

*1 Haskell では `data Num = One | Two | Three | Many` と書く。

*2 Haskell では `data Num = One | Two | Three | Many deriving Eq` と書く。

*3 Haskell では `data Rectangle = Rectangle Float Float Float Float` と書く。

これは C で言う `struct` と似た用法である。^{*4}

Rectangle 型から中身を取り出すには、次のような関数を用意しておかねばならない。

$$x \text{ Rectangle } \llbracket a _ _ _ \rrbracket = a \quad (11.6)$$

$$y \text{ Rectangle } \llbracket _ a _ _ \rrbracket = a \quad (11.7)$$

$$\text{width} \text{ Rectangle } \llbracket _ _ a _ \rrbracket = a \quad (11.8)$$

$$\text{height} \text{ Rectangle } \llbracket _ _ _ a \rrbracket = a \quad (11.9)$$

この面倒は、次のシンタックスシュガーを使うことで軽減される。^{*5}

型 **Rectangle** のようにパラメタが多いときは、パラメタに名前があると便利である。そこで利用できるのがレコード構文である。レコード構文を使うと、式 (11.4) は

$$\begin{aligned} &\text{datatype } \text{Rectangle} \\ &= \text{Rectangle } \{x :: \text{Float}, y :: \text{Float}, \text{width} :: \text{Float}, \text{height} :: \text{Float}\} \\ &\quad \text{deriving } \text{Eq} \quad (11.10) \end{aligned}$$

のように書き直すことができる。^{*6}

^{*4} Haskell では

```
r :: Rectangle
r = Rectangle 1 2 3 4
```

と書く。

^{*5} Haskell では

```
x      Rectangle a _ _ _ = a
y      Rectangle _ a _ _ = a
width  Rectangle _ _ a _ = a
height Rectangle _ _ _ a = a
```

と書く。

^{*6} Haskell では

```
data Rectangle = Rectangle {
```

レコード構文を用いると、型から中身を取り出す関数は自動的に定義される。また

$$r = \text{Rectangle } \{x = 1, y = 2, \text{width} = 3, \text{height} = 4\} \quad (11.11)$$

のようなレコード構文専用の初期化を行っても良いし、従来の初期化方法を用いても良い。^{*7}

* * *

データ型の定義には、複数の値コンストラクタを指定できる。そのため

$$\begin{aligned} \text{datatype } \text{Shape} = & \text{Circle } [\text{Float Float}] \\ & \vee \text{Triangle } [\text{Float Float Float}] \end{aligned} \quad (11.12)$$

のようなデータ型の定義も可能である。この例では型 **Shape** は値コンストラクタ **Circle** または **Triangle** によって初期化され、それぞれ 2 個または 3 個の **Float** 型のパラメータを取る。^{*8}

11.3 型クラスとインスタンス化

型クラスとは、複数の型が持つ共通のインタフェースである。その複数の型をいま **a** で表すこととして、型 **a** が型クラス **Eq** に属すとしよう。型ク

```
x      :: Float,
y      :: Float,
width  :: Float,
height :: Float
}
deriving Eq
```

と書く。

^{*7} Haskell では `r = Rectangle { x = 1, y = 2, width = 3, height = 4 }` と書く。

^{*8} Haskell では `data Shape = Circle Float Float | Triangle Float Float Float` と書く。

ラス **Eq** は等号 (\equiv) と不等号 (\neq) をインタフェースとして持つ. このことを

$$\text{typeclass } \mathbf{Eq} \supset \mathbf{a} \text{ where } \left\{ \begin{array}{l} (\equiv) :: \mathbf{a} \rightarrow \mathbf{a} \rightarrow \mathbf{Bool} \\ (\neq) :: \mathbf{a} \rightarrow \mathbf{a} \rightarrow \mathbf{Bool} \\ x \equiv y = \neg(x \neq y) \\ x \neq y = \neg(x \equiv y) \end{array} \right. \quad (11.13)$$

と書く.*9

TK: typeclass による具象型の型クラス宣言.

ここで宣言したのは等号, 不等号というインタフェースが「ある」という事だけで, その実装は未定義である. 等号, 不等号の実装は次に述べる instance 文を使う.

型クラス **Eq** に属す型 **a** は等号と不等号を持つ. 我々の型 **Num** が型クラス **Eq** に属すことを宣言し, 型クラス **Eq** が備えるべき等号, 不等号を実装するには

$$\text{instance } \mathbf{Eq} \supset \mathbf{Num} \text{ where } \left\{ \begin{array}{l} \mathbf{One} \equiv \mathbf{One} = \mathbf{True} \\ \mathbf{Two} \equiv \mathbf{Two} = \mathbf{True} \\ \mathbf{Three} \equiv \mathbf{Three} = \mathbf{True} \\ \mathbf{Many} \equiv \mathbf{Many} = \mathbf{True} \\ _ \equiv _ = \mathbf{False} \end{array} \right. \quad (11.14)$$

とする. これを**型クラスのインスタンス化**と呼ぶ.*10

TK: instance による具象型のインスタンス宣言.

*9 Haskell では typeclass を class と書く. また \supset を省略して,

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x==y = not(x/=y)
  x/=y = not(x==y)
```

と書く.

*10 Haskell では

型クラスは継承関係を持てる．型クラス *Ord* は型クラス *Eq* から等号，不等号を継承し「小なりイコール」 (\leq) を追加する．

`typeclass Eq ⊃ a ⇒ Ord ⊃ a where (≤) :: a → a → Bool` (11.15)

型クラス *Ord* に属する型は，等号，不等号，小なりイコールと，それらから派生させることのできる大なり ($>$)，大なりイコール (\geq)，小なり ($<$)，および最大値をとる関数 `max` と最小値をとる関数 `min` を持つ．^{*11}

* * *

型は複数の型クラスに同時に属することができる．例えば型 *Num* は型クラス *Eq* と同時に型クラス *Ord* に属することもできる．それには

`datatype Num = One ∨ Two ∨ Three ∨ Many deriving (Eq, Ord)`
(11.16)

とする．^{*12}

```
instance Eq Num where
  One==One    = True
  Two==Two    = True
  Three==Three = True
  Many==Many = True
  _==_        = False
```

と書く．

^{*11} Haskell では

```
class (Eq a) => Ord a where (≤) :: a -> a -> Bool
```

と書く．

^{*12} Haskell では

```
data Num = One | Two | Three | Many deriving (Eq, Ord)
```

と書く．

型クラス **Ord** のインスタンスは小なりイコール演算子 (\leq) が定義されていなければならない。我々は

$$\text{instance } \mathbf{Ord} \supset \mathbf{Num} \text{ where } \left\{ \begin{array}{l} \mathbf{One} \leq \mathbf{One} = \mathbf{True} \\ \mathbf{One} \leq \mathbf{Two} = \mathbf{True} \\ \mathbf{One} \leq \mathbf{Three} = \mathbf{True} \\ \mathbf{One} \leq \mathbf{Many} = \mathbf{True} \\ \mathbf{Two} \leq \mathbf{Two} = \mathbf{True} \\ \mathbf{Two} \leq \mathbf{Three} = \mathbf{True} \\ \mathbf{Two} \leq \mathbf{Many} = \mathbf{True} \\ \mathbf{Three} \leq \mathbf{Three} = \mathbf{True} \\ \mathbf{Three} \leq \mathbf{Many} = \mathbf{True} \\ \mathbf{Many} \leq \mathbf{Many} = \mathbf{True} \\ _ \leq _ = \mathbf{False} \end{array} \right. \quad (11.17)$$

のように **Num** の \leq 演算子を定義することができる。

11.4 余談：レンズ*

~~TK: Lens~~

~~TK: Move to 12.4~~

データ型にはシノニム（別名）がつけられる。例えば **Char** のリスト **[Char]** は

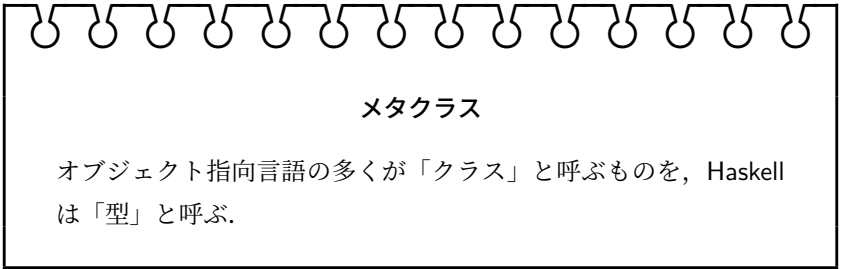
$$\text{typesynonym } \mathbf{String} = [\mathbf{Char}] \quad (11.18)$$

とすることで、別名 **String** を与えることができる。Haskell では typesynonym を type と省略してしまう。これは残念なことだが、C の typedef のようなものだと思って割り切るしかない。^{*13}

^{*13} Haskell では `type String = [Char]` と書く。

11.5 この章のまとめ*

1. ...



メタクラス

オブジェクト指向言語の多くが「クラス」と呼ぶものを, Haskell
は「型」と呼ぶ.

第 12 章

多相型の定義*

...

12.1 多相型

Maybe のように型パラメタを取る型を多相型と呼ぶ。多相型はどのように定義されるかと言うと、

$$\text{datatype}^\dagger \text{ Maybe } a = \text{Just} \llbracket a \rrbracket \vee \emptyset \quad (12.1)$$

のように、やはり `datatype` を使って定義される。念のため型パラメタを取る場合は `datatype†` と区別しておこう。^{*1}

Maybe が型クラス `Eq` に属すものとして、`Eq` からのインスタンス化をしておこう。ここでも型パラメタ付きの `instance` を仮に `instance†` とすると次のように書けそうである。

$$\text{instance}^\dagger \text{ Eq } \sqsupset (\text{Maybe } a) \text{ where } \begin{cases} \text{Just} \llbracket x \rrbracket \equiv \text{Just} \llbracket y \rrbracket = x \equiv y \\ \emptyset \equiv \emptyset = \text{True} \\ _ \equiv _ = \text{False} \end{cases} \quad (12.2)$$

^{*1} Haskell は `datatype` と `datatype†` を区別せず、`datatype† Maybe a = Just ⌊a⌋ ∨ ∅` を `data Maybe a = Just a | Nothing` と書く。

残念ながら、この式は **a** 型の変数 x, y の間に等号 (\equiv) が定義されていることが隠れた前提になっているため、正しくない。我々は **a** が型クラス **Eq** に属することを要求するので、次のように言い換える。

$$\text{instance}^\dagger \text{Eq} \supset \mathbf{a} \Rightarrow \text{Eq} \supset (\text{Maybe } \mathbf{a})$$

$$\text{where} \begin{cases} \text{Just } \llbracket x \rrbracket \equiv \text{Just } \llbracket y \rrbracket = x \equiv y \\ \emptyset \equiv \emptyset = \text{True} \\ _ \equiv _ = \text{False} \end{cases} \quad (12.3)$$

この $\text{Eq} \supset \mathbf{a} \Rightarrow$ の部分が「以下 **a** 型は **Eq** 型クラスに属するものとして」という意味になる。^{*2}

Maybe の間に新たに定義された等号 (\equiv) は次の型を持つ。

$$(\equiv) :: \text{Eq} \supset \mathbf{a} \Rightarrow \text{Maybe } \llbracket \mathbf{a} \rrbracket \rightarrow \text{Maybe } \llbracket \mathbf{a} \rrbracket \rightarrow \text{Bool} \quad (12.4)$$

* * *

多相型を定義するとき、型パラメタは 2 個以上与えられても良い。例えば **Either** は次のように定義できる。

$$\text{datatype}^\dagger \text{Either } \mathbf{a} \mathbf{b} = \text{Left } \llbracket \mathbf{a} \rrbracket \vee \text{Right } \llbracket \mathbf{b} \rrbracket \quad (12.5)$$

12.2 自己参照型

型の定義中に自分自身を参照する型を自己参照型または再帰型と呼ぶ。例えばリストは

$$\text{datatype}^\dagger \text{List } \mathbf{a} = [] \vee \mathbf{a} : \text{List } \mathbf{a} \quad (12.6)$$

^{*2} Haskell では

```
instance Eq a => Eq (Maybe a) where
  Just x == Just y = x==y
  Nothing==Nothing = True
  _==_              = False
```

と書く。

のように定義できる.

12.3 関手の拡張

TK: Writing

$$\text{typeclass}^\dagger \text{ *Functor* } \supset \text{ *f* where } (\cdot) :: (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow^f \llbracket \mathbf{a} \rrbracket \rightarrow^f \llbracket \mathbf{b} \rrbracket \quad (12.7)$$

TK: `typeclass†` による型コンストラクタの型クラス宣言.

型クラス *Functor* はマップ演算子 (\cdot) を提供する. リスト型は *Functor* 型クラスのインスタンスなので,

$$\text{instance}^\dagger \text{ *Functor* } \supset \text{ *List* where } (\cdot) = (\odot) \quad (12.8)$$

と定義する.*3

TK: `instance†` による型コンストラクタのインスタンス宣言.

TK: Follow: That's it! Notice how we didn't write `instance Functor [a] where`, because from `fmap :: (a → b) → f a → f b`, we see that the `f` has to be a type constructor that takes one type. `[a]` is already a concrete type (of a list with any type inside it), while `[]` is a type constructor that takes one type and can produce types such as `[Int]`, `[String]` or even `[[String]]`.

Maybe 型の場合は

$$\text{instance}^\dagger \text{ *Functor* } \supset \text{ *Maybe* where } \begin{cases} f \cdot \text{Just } \llbracket x \rrbracket = \text{Just } \llbracket f x \rrbracket \\ f \cdot \emptyset = \emptyset \end{cases} \quad (12.9)$$

となる.*4

*3 Haskell では

```
instance Functor [] where fmap = map
```

と書く.

*4 Haskell では

Either 型の場合は

$$\text{instance}^\dagger \text{ Functor } \supset (\text{Either } a) \text{ where } \begin{cases} f \cdot \text{Right} \llbracket x \rrbracket = \text{Right} \llbracket f x \rrbracket \\ f \cdot \text{Left} \llbracket x \rrbracket = \text{Left} \llbracket x \rrbracket \end{cases} \quad (12.10)$$

となる.*5

TK: Multiple parameter extension

12.4 余談: newtype*

リストに新しいマップ演算子 \circledast を定義したいとしよう. この演算子 \circledast は

$$[f, g, h] \circledast [x, y, z] = [f x, g y, h z] \quad (12.11)$$

のように働くとする. 新しいマップ演算子 \circledast のことを我々はジップ演算子と呼ぶことにする.

ジップ演算子を定義するにはどうしたら良いだろうか. 汎用性を考えると, ジップ演算子もまたリストのアプリカティブマップ演算子であって欲しい. ところが, リストには既に \otimes というアプリカティブマップ演算子が定義されている. 念のためにアプリカティブマップ演算子を用いると

$$[f, g, h] \otimes [x, y, z] = [f x, f y, f z, g x, g y, g z, h x, h y, h z] \quad (12.12)$$

```
instance Functor Maybe where
  fmap f (Maybe x) = Maybe (f x)
  fmap f Nothing = Nothing
```

と書く.

*5 Haskell では

```
instance Functor (Either a) where
  fmap f (Right x) = Right (f x)
  fmap f (Left x) = Left x
```

と書く.

である．ある型のインスタンス化は一種類しか行えないので，リスト型に新しいアプリカティブマップ演算子を追加することは出来ない．

そこで `datatype†` を使ってリスト型をラップした新しい型を作る．例えば新しい型を $\text{ZL}[\mathbf{a}]$ とすると

$$\text{datatype}^{\dagger} \text{ ZipList } \mathbf{a} = \text{ZipList } \{\text{getList} :: [\mathbf{a}]\} \quad (12.13)$$

のように定義することになる．念のため，左辺の **ZipList** は型コンストラクタ，右辺の `ZipList` は値コンストラクタである．すなわち

$$\text{ZL}[\mathbf{a}] = \text{ZipList } \mathbf{a} \quad (12.14)$$

である．

こうしておいて，あとは

$$\begin{aligned} \text{instance}^{\dagger} \text{ Functor } \supset \text{ ZipList} \\ \text{where } f \cdot \text{ZipList } [x_s] = \text{ZipList } [f \odot x_s] \end{aligned} \quad (12.15)$$

および

$$\begin{aligned} \text{instance}^{\dagger} \text{ Applicative } \supset \text{ ZipList} \\ \text{where } \begin{cases} [x] = \text{ZipList } [\text{repeat } x] \\ \text{ZipList } [f_s] \times \text{ZipList } [x_s] = \text{ZipList } [f_s \otimes x_s] \end{cases} \end{aligned} \quad (12.16)$$

と定義する．ここで

$$\begin{cases} (\otimes) \cdot [] = [] \\ (\otimes) [] \cdot = [] \\ (\otimes)(x : x_s)(y : y_s) = (x \$ y) : x_s \otimes y_s \end{cases} \quad (12.17)$$

であると定義する．

関数 `repeat` は引数を無限回繰り返すリストを返す関数である．念のため関数 `repeat` の実装方法を書いておくと

$$\begin{aligned} & \text{repeat} :: [\mathbf{a}] \rightarrow [\mathbf{a}] \\ & \text{repeat } x = x : \text{repeat } x \end{aligned} \quad (12.18)$$

である.

任意のリスト x_s と任意の関数リスト f_s ただし

$$x_s = [x_0, x_1, \dots] \quad (12.19)$$

$$f_s = [f_0, f, \dots] \quad (12.20)$$

の両方をそれぞれ値コンストラクタで包んでアプリカティブマップ演算子を適用すると

$$\text{ZipList} \llbracket f_s \rrbracket \times \text{ZipList} \llbracket x_s \rrbracket = [f_0 x_0, f x_1, \dots] \quad (12.21)$$

のようにジップ演算子を適用できる.

このままでも問題は無いのだが, 式 (12.13) を次のように書き換えることがより好ましい.

$$\text{newtype}^\dagger \text{ ZipList } a = \text{ZipList} \{ \text{getList} :: [a] \} \quad (12.22)$$

やったことはキーワード `datatype†` を `newtype†` に置き換えたことである. これは, 字面に反して, 型 $\text{ZL} \llbracket a \rrbracket$ が何一つ「新しくない」ことを Haskell コンパイラに伝えるためである. その結果 Haskell コンパイラは型 $\text{ZL} \llbracket a \rrbracket$ に対する最適化の機会を得る.

* * *

式 (12.17) にもう一段の抽象化をしておこう. 関数 `zipWith` を

$$\left\| \begin{array}{l} \text{zipWith} :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c] \\ \left\{ \begin{array}{l} \text{zipWith } [] = [] \\ \text{zipWith } _ [] = [] \\ \text{zipWith } f (x : x_s) (y : y_s) = fxy : \text{zipWith } f x_s y_s \end{array} \right. \end{array} \right. \quad (12.23)$$

と定義する. こうすれば, 演算子 \circledast は

$$(\circledast) = \text{zipWith}(\$) \quad (12.24)$$

と関数 `zipWith` から定義できる.

関数 `zipWith` を再帰的に定義したが、再帰をリストのマッピング演算子に押し込むこともできる。それには

$$\text{zipWith } f(x : x_s)(y : y_s) = f' \odot z_s \text{ where } \begin{cases} f' \triangleq \text{uncurry } f \\ z_s \triangleq \text{zip } x_s y_s \end{cases} \quad (12.25)$$

と、先に `zip` 関数を定義しておいて、その後 `zipWith` 関数を定義する。ここに `zip` 関数は

$$\begin{cases} \text{zip} :: [a] \rightarrow [b] \rightarrow ([a], [b]) \\ \text{zip } xy = (x, y) \end{cases} \quad (12.26)$$

であり、`uncurry` 関数は

$$\begin{cases} \text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c \\ \text{uncurry } fxy = f(x, y) \end{cases} \quad (12.27)$$

である。^{*6}

12.5 この章のまとめ*

1. ...

^{*6} Haskell では `zipWith` 関数を

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f' 'map' zs where
    f' = uncurry f
    zs = zip xs ys
```

と書ける。ただし `zipWith` は `Prelude` から提供される。

第 13 章

カテゴリー*

13.1 多値*

Haskell の関数の引数は常にひとつであり，戻り値も常にひとつである．しかし，複数の戻り値を返したい場合もある．例えば実数 x の平方根は \sqrt{x} および $-\sqrt{x}$ のふたつである．そこで次のような関数 `sqrts` ただし

$$\left\| \begin{array}{l} \text{sqrts} :: \text{Real} \supset \mathbf{a} \Rightarrow \mathbf{a} \rightarrow [\mathbf{a}] \\ \text{sqrts } x = [\text{sqrt } x, (-\text{sqrt } x)] \end{array} \right. \quad (13.1)$$

を考えてみる.*¹

ここで型 $\mathbf{a} \rightarrow [\mathbf{a}]$ に別名を与えよう．別名を与えるには型シノニムを定義する方法と，`newtype` を用いる方法とがある．前者は単に読みやすさの向上のためだけであるが，後者は第 12 章で見たように特定の型クラスの新しいインスタンスとしてその型を定義するためである．我々は後者を選択するこ

*¹ Haskell では

```
sqrts :: Real a => a -> [a]
sqrts = [sqrt x, (-sqrt x)]
```

と書く．

とにして

$$\text{newtype } \textit{NonDet} \textit{ a b} = \text{NonDet} \{ \text{run} :: \textit{a} \rightarrow [\textit{b}] \} \quad (13.2)$$

とする.*2

型コンストラクタ *NonDet* によって作られる型 $\text{NonDet} \llbracket \textit{a b} \rrbracket$ は型 $\textit{a} \rightarrow [\textit{a}]$ とは異なる型であるため、関数 *sqrts* の新しいバージョンが必要になる。そこで

$$\left\| \begin{array}{l} \text{sqrts}' :: \textit{Real} \supset \textit{a} \Rightarrow \text{NonDet} \llbracket \textit{a a} \rrbracket \\ \text{sqrts}' = \text{NonDet} \llbracket \text{sqrts} \rrbracket \end{array} \right. \quad (13.3)$$

のように定義しよう.*3

関数 *sqrts'* はコンテナ $\text{NonDet} \llbracket \dots \rrbracket$ に包まれているので、

$$y = \text{run } \text{sqrts}' \ x \quad (13.4)$$

のようにして呼び出さねばならない.*4

次節から、多値を返す関数の適用と合成を一般化する。その前に用語の説明をしておこう。複数の値を返す関数のうち、返す値の数が定まらないような関数を**非決定的な関数**と呼ぶ。リストは一般に長さが定まらないので、リストを返す関数は非決定的な関数である。

*2 Haskell では

```
newtype NonDet a b = NonDet { run :: a -> [b] }
```

と書く。

*3 Haskell では

```
sqrts' :: Real a => NonDet a a
sqrts' = NonDet sqrts
```

と書く。

*4 Haskell では

```
y = run sqrts' x
```

と書く。

13.2 カテゴリー*

TK: Writing

Functor は一般マップ演算子 (\cdot) を共通のインタフェースとして持つ型を抽象化した型クラス。インスタンスであるリストの実装は \odot を使う。

非決定的な関数 `duplicate` と `triplicate` を

$$\begin{array}{|l} \text{duplicate, triplicate} :: \mathbf{a} \rightarrow [\mathbf{a}] \\ \text{duplicate } x = [x, x] \\ \text{triplicate } x = [x, x, x] \end{array} \quad (13.5)$$

のように定義しよう。^{*5}

このふたつの関数 `duplicate` と `triplicate` をもし合成できるとしたら、合成された関数は `x` を引数に取り `[x, x, x, x, x, x]` を返すものとするのが自然であろう。しかし、どのように合成したら良いだろうか。

いま欲しいのは、

$$y = (\text{triplicate} \square \text{duplicate})x \quad (13.6)$$

としたときに $y = [x, x, x, x, x, x]$ となるような合成演算子 \square である。ここで関数合成演算子 (\cdot) が使えないことは明らかである。と言うのも、関数 `triplicate` はリストを引数に取らないため

$$\text{triplicate} \cdot \text{duplicate } x = \text{triplicate}[x, x] \dots \text{型エラー!} \quad (13.7)$$

^{*5} Haskell では

```
duplicate, triplicate :: a -> [a]
duplicate x = [x, x]
triplicate x = [x, x, x]
```

と書く。

となってしまうからである。リストのマッピング演算子 (\odot) を使うと

$$\text{triplicate} \odot (\text{duplicate } x) = \text{triplicate} \odot [x, x] \quad (13.8)$$

$$= [[x, x, x], [x, x, x]] \quad (13.9)$$

であるから、型エラーは回避できる。

このように検討していくと、このふたつの関数 `duplicate` と `triplicate` は

$$y = \flat(\text{triplicate} \odot (\text{duplicate } x)) \quad (13.10)$$

のように合成できることがわかる。これを関数合成演算子を使って分解すると

$$y = \flat.(\text{triplicate} \odot). \text{duplicate} \$ x \quad (13.11)$$

となり、少しは読みやすくなる。しかし、合成のたびに平坦化演算子 (\flat) やマッピング演算子 (\odot) を書くのは煩雑であるし、3 段以上の合成になると手に負えなくなる。

そこで、関数合成をオーバーライドする方法が欲しくなる。関手のマッピング演算子 (\cdot) は各々のインスタンスで独自にオーバーライドされていた。リストのマッピング演算子 (\odot) や `Maybe` のマッピング演算子 (\sqcup) がそれらである。同じように、関数合成 (\cdot) も一般化したものが欲しい。

それがカテゴリの合成演算子 (\bullet) である。カテゴリは *Category* で表される型クラスである。

カテゴリ型クラスの定義に立ち入る前に、新しい表記を導入しておこう。これまで型 **a** の引数をひとつ取り、型 **b** の戻り値を返す関数の型を $\mathbf{a} \rightarrow \mathbf{b}$ または $(\rightarrow)\mathbf{ab}$ と書いてきた。これから導入する型は、やはり型 **a** の引数を取り、型 **b** の戻り値を返す。この新しい型を $\mathbf{a} \xrightarrow{c} \mathbf{b}$ または $(\xrightarrow{c})\mathbf{ab}$ と書こう。ここに **c** は型コンストラクタであり、カテゴリ型クラスのインスタンスである。型 $(\xrightarrow{c})\mathbf{ab}$ の値は関数ではなく射と呼ぶ。^{*6}

^{*6} Haskell では $(\xrightarrow{c})\mathbf{ab}$ を $\mathbf{c} \mathbf{a} \mathbf{b}$ と書く。

カテゴリ型クラス (*Category*) は次のように定義されている。

$$\text{typeclass}^\dagger \text{ Category} \supset c \text{ where } \begin{cases} \text{id} :: x \xrightarrow{c} y \\ (\bullet) :: (x \xrightarrow{c} y) \rightarrow (x \xrightarrow{c} y) \rightarrow (x \xrightarrow{c} y) \end{cases} \quad (13.12)$$

—

$$\xrightarrow{\text{NonDet}} = a \rightarrow [a] \quad (13.13)$$

としよう.*7

—

$$\text{id} :: a \xrightarrow{c} b \quad (13.14)$$

$$(\bullet) :: (b \xrightarrow{c} c) \rightarrow (a \xrightarrow{c} b) \rightarrow (a \xrightarrow{c} c) \quad (13.15)$$

e.g.

$$a \xrightarrow{c} b = a \rightarrow [a] \quad (13.16)$$

—

このような問題を解決するのがカテゴリすなわち *Category* 型クラスである。

Category 型クラスは次のように定義されている。

$$\text{typeclass}^\dagger \text{ Category} \supset c \text{ where } \begin{cases} \text{id} :: c \ x \ x \\ (\bullet) :: c \ x \ y \rightarrow c \ x \ y \rightarrow c \ x \ y \end{cases} \quad (13.17)$$

*7 Haskell では

```
newtype NonDet = NonDet a -> [a]
instance Category NonDet where ...
```

と書く。... の部分は後述する。

型パラメタ \mathbf{c} が $\mathbf{c} = (\rightarrow)$ のとき $\mathbf{id} = \mathbf{id}$ かつ $\bullet = \cdot$ になるので、圏は関数を何やら拡張したものであることがわかる。

我々は $\mathbf{duplicate}$ や $\mathbf{triplicate}$ の型を考えて

$$\mathbf{c} \mathbf{x} \mathbf{y} = \mathbf{x} \rightarrow [\mathbf{y}] \quad (13.18)$$

と考えてみよう。つまり $\mathbf{c} = ((\rightarrow) \diamond [\diamond])$ と考えるわけである。

そうすると、まず \mathbf{id} は $\mathbf{a} \rightarrow [\mathbf{a}]$ 型でなければならないから

$$\mathbf{id} = \lambda x \mapsto [x] \quad (13.19)$$

となる。

次に (\bullet) であるが、 $\mathbf{x} = \mathbf{y}$ とすると、型は $(\mathbf{x} \rightarrow [\mathbf{x}]) \rightarrow (\mathbf{x} \rightarrow [\mathbf{x}]) \rightarrow (\mathbf{x} \rightarrow [\mathbf{x}])$ であるから、式 (13.11) から

$$\mathbf{triplicate} \bullet \mathbf{duplicate} = \mathbf{b} \cdot (\mathbf{triplicate} \odot) \cdot \mathbf{duplicate} \quad (13.20)$$

をそのまま抽象化して

$$f \bullet g = \mathbf{b} \cdot (f \odot) \cdot g \quad (13.21)$$

とすれば、そのまま使えそうである。

これで、我々の型コンストラクタ \mathbf{NonDet} を $\mathbf{Category}$ 型クラスのインスタンスにできる。ただし、式 (13.19) ならびに式 (13.21) は型 $\mathbf{NonDet} \llbracket \mathbf{a} \mathbf{b} \rrbracket$ に対応していないので、次のように値コンストラクタで包んだ上でインスタンス化を行う。

$$\begin{aligned} &\text{instance}^\dagger \mathbf{Category} \supset \mathbf{NonDet} \\ &\text{where} \left\{ \begin{array}{l} \mathbf{id} = \mathbf{NonDet} \llbracket \lambda x \mapsto [x] \rrbracket \\ \mathbf{NonDet} \llbracket f \rrbracket \bullet \mathbf{NonDet} \llbracket g \rrbracket = \mathbf{NonDet} \llbracket \mathbf{b} \cdot (f \odot) \cdot g \rrbracket \end{array} \right. \quad (13.22) \end{aligned}$$

続けて、 $\text{NonDet}[\![\mathbf{a\,b}]\!]$ 型バージョンの `duplicate` と `triplicate` を定義しておこう。

$$\begin{array}{l} \parallel \text{duplicate}' :: \text{NonDet}[\![\mathbf{a\,a}]\!] \\ \parallel \text{duplicate}' = \text{NonDet}[\![\text{duplicate}]\!] \end{array} \quad (13.23)$$

$$\begin{array}{l} \parallel \text{triplicate}' :: \text{NonDet}[\![\mathbf{a\,a\,a}]\!] \\ \parallel \text{triplicate}' = \text{NonDet}[\![\text{triplicate}]\!] \end{array} \quad (13.24)$$

これで、式 (13.11) は

$$y = \text{run}(\text{triplicate}' \bullet \text{duplicate}')x \quad (13.25)$$

のようにシンプルに書き下すことができる。念のため右辺の括弧の中身を展開しておく

$$\text{triplicate}' \bullet \text{duplicate}' = \text{NonDet}[\![\text{triplicate}]\!] \bullet \text{NonDet}[\![\text{duplicate}]\!] \quad (13.26)$$

$$= \text{NonDet}[\![\mathbf{\triangleright} \cdot (\text{triplicate} \odot) \cdot \text{duplicate}]\!]\quad (13.27)$$

である。

* * *

Category 型クラスは `Control.Category` で

```
class Category c where
  id  :: c x x
  (.) :: c y z -> c x y -> c x z
```

Haskell

と定義されており、`Prelude` の `id` および `.` と名前が衝突している。従って `Category` 型クラスの `id` および `.` を使用する際は

```
import Control.Category as Cat
```

Haskell

としておき、`Cat.id` および `Cat..` として使用する。なお `Cat..` は `<<<` とも定義されている。

インスタンス化のことまで考えると、次のようにインポートしインスタス化するのが良さだろう。

Haskell

```
-- duplicate-triplicate.hs
import qualified Control.Category as Cat

newtype NonDet a b = NonDet { run :: a -> [b] }

instance Cat.Category NonDet where
    id          = NonDet (\x -> [x])
    (NonDet f) . (NonDet g) = NonDet (concat . map f . g)

duplicate :: a -> [a]
duplicate x = [x, x]

triplicate :: a -> [a]
triplicate x = [x, x, x]

duplicate' :: NonDet a a
duplicate' = NonDet duplicate

triplicate' :: NonDet a a
triplicate' = NonDet triplicate

x = "Hello."
y = run (triplicate' Cat.. duplicate') x
```

ところで、リストのバインド演算子 (\clubsuit) は

$$f \clubsuit x_s = \flat(f \odot x_s) \quad (13.28)$$

であった。関数 f, g が $\text{NonDet} \llbracket f \rrbracket, \text{NonDet} \llbracket g \rrbracket$ として与えられることを考えると、関数 f, g の型は $a \rightarrow [a]$ で確定するので、式 (13.28) 中のバインド演算子 (\heartsuit) はリストのバインド演算子 (\clubsuit) で確定する。

そこでリストのバインド演算子を用いると

$$\flat \cdot (f \odot) \cdot g = \lambda x \mapsto \flat (f \odot (gx)) \quad (13.29)$$

$$= \lambda x \mapsto f \clubsuit (gx) \quad (13.30)$$

$$= f \boxtimes g \quad (13.31)$$

であるから、式 (13.22) の 2 行目は

$$\text{NonDet} \llbracket f \rrbracket \bullet \text{NonDet} \llbracket g \rrbracket = \text{NonDet} \llbracket f \boxtimes g \rrbracket \quad (13.32)$$

とも書ける。改めて式 (13.22) を書き直すと

$$\begin{aligned} & \text{instance}^\dagger \text{Category} \supset \text{NonDet} \\ & \text{where} \begin{cases} \text{id } x = \text{NonDet} \llbracket x \rrbracket \\ \text{NonDet} \llbracket f \rrbracket \bullet \text{NonDet} \llbracket g \rrbracket = \text{NonDet} \llbracket f \boxtimes g \rrbracket \end{cases} \end{aligned} \quad (13.33)$$

である。Haskell で書けば

```
instance Cat.Category NonDet where
  id          = NonDet pure
  (NonDet f) . (NonDet g) = NonDet (f <=< g)
```

Haskell

である。

13.3 カテゴリー則*

$$\text{id} \bullet f = f \bullet \text{id} = f \quad (13.34)$$

$$(f \bullet g) \bullet h = f \bullet (g \bullet h) \quad (13.35)$$

13.4 余談: Kleisli 型*

$$\text{newtype } \text{ItCouldBe } \mathbf{a} \mathbf{b} = \text{ItCouldBe} \{ \text{runIt} :: \mathbf{a} \rightarrow \text{Maybe} \llbracket \mathbf{b} \rrbracket \} \quad (13.36)$$

$$\left\| \begin{array}{l} \text{rec} :: (\textcolor{red}{Real} \supset \mathbf{a}) \Rightarrow \mathbf{a} \rightarrow \mathbf{a} \\ \text{rec } x \mid_{x \neq 0} = \text{Just} \llbracket 1/x \rrbracket \\ \quad \mid_{\text{otherwise}} = \emptyset \end{array} \right\| \quad (13.37)$$

$$y = \sharp.(f \sqcap).g \$ x \quad (13.38)$$

$$\text{rec}' = \text{ItCouldBe} \llbracket \text{rec} \rrbracket \quad (13.39)$$

—

$$\text{newtype } \textcolor{red}{NonDet} \mathbf{a} \mathbf{b} = \text{NonDet} \llbracket \mathbf{a} \rightarrow \llbracket \mathbf{b} \rrbracket \rrbracket \quad (13.40)$$

$$\text{newtype } \textcolor{red}{ItCouldBe} \mathbf{a} \mathbf{b} = \text{ItCouldBe} \llbracket \mathbf{a} \rightarrow \text{Maybe} \llbracket \mathbf{b} \rrbracket \rrbracket \quad (13.41)$$

$$\text{newtype } \textcolor{red}{Kleisli} \, m \, \mathbf{a} \, \mathbf{b} = \text{Kleisli} \llbracket \mathbf{a} \rightarrow^m \llbracket \mathbf{b} \rrbracket \rrbracket \quad (13.42)$$

$$\text{typesynonym } \textcolor{red}{ItCouldBe} \mathbf{a} \mathbf{b} = \textcolor{red}{Kleisli} \mathbf{a} \mathbf{b} \quad (13.43)$$

第 14 章

アロー*

カテゴリが抽象化したのは関数の合成すなわち関数の分解方法である。モナドもカテゴリの一種である。ただし関数 g の値が関数 f の引数に依存する場合は $g \bullet f$ のように両者を合成できない。そこで考えられたのがアローである。

14.1 アロー*

数値のリスト x_s の平均をとる関数 `mean` を考えてみよう。実装は単純で

$$\text{mean } x_s = \text{sum } x_s / \text{length } x_s \quad (14.1)$$

である。

問題はここからである。関数 `mean` を関数 `sum` と関数 `length` の合成の形にできるだろうか。つまり

$$\text{mean } x_s = (\text{sum} \square \text{length}) x_s \quad (14.2)$$

あるいは

$$\text{mean} = \text{sum} \square \text{length} \quad (14.3)$$

の形に分解するような関数合成演算子は存在するだろうか。

残念なことに関数 `sum` も `length` も同じ引数 (x_s) をとる必要がある。そのため従来の方法では関数を合成出来ない。しかし次のように演算の結果をペアに収めることが出来れば何とかなりそうである。

$$\text{mean } x_s = \left(\backslash \left(\begin{array}{c} x \\ y \end{array} \right) \mapsto x/y \right) \left(\begin{array}{c} \text{sum } x_s \\ \text{length } x_s \end{array} \right) \quad (14.4)$$

幸いカーリー化された 2 引数関数をペアを引数にとる関数へ変換する `uncurry` 関数があり、

$$\text{uncurry}(\star) = \backslash \left(\begin{array}{c} x \\ y \end{array} \right) \mapsto x \star y \quad (14.5)$$

であるから、式 (14.4) の第 1 項（前半の括弧）は `uncurry(/)` と出来る。

式 (14.4) の第 2 項（後半の括弧）は

$$\left(\begin{array}{c} \text{sum } x_s \\ \text{length } x_s \end{array} \right) = \left[\begin{array}{c} \text{sum} \\ \text{length} \end{array} \right] \left(\begin{array}{c} x_s \\ x_s \end{array} \right) \quad (14.6)$$

という形にしたい。そうすれば

$$\text{split } x = \left(\begin{array}{c} x \\ x \end{array} \right) \quad (14.7)$$

なる関数 `split` を用意すると、後は新しく導入したブラケット $\left[\begin{array}{c} \vdots \end{array} \right]$ だけの問題になる。

問題を整理すると、式 (14.4) を

$$\text{mean } x_s = \text{uncurry}(/) \cdot \left[\begin{array}{c} \text{sum} \\ \text{length} \end{array} \right] \cdot \text{split } \$ x_s \quad (14.8)$$

という形に持っていきたい。それには

$$\left(\begin{array}{c} fx \\ gy \end{array} \right) = \left[\begin{array}{c} f \\ g \end{array} \right] \left(\begin{array}{c} x \\ y \end{array} \right) \quad (14.9)$$

となるような関数 f と関数 g の合成 $\left[\begin{array}{c} \vdots \end{array} \right]$ が是非とも必要である。この特殊な関数合成を仮にブラケットと呼ぶことにしよう。

そして、関数の合成を考える以上

$$\begin{bmatrix} f \cdot f' \\ g \cdot g' \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} \bullet \begin{bmatrix} f' \\ g' \end{bmatrix} \quad (14.10)$$

なる合成演算子 (\bullet) も同時に考えておく必要がある。と言うのも、ブラケットの合成が自由にできるようになると

$$\begin{pmatrix} \textcolor{blue}{f} \textcolor{blue}{x} \\ \textcolor{green}{y} \end{pmatrix} = \begin{bmatrix} f \\ \text{id} \end{bmatrix} \begin{pmatrix} \textcolor{blue}{x} \\ \textcolor{blue}{y} \end{pmatrix} \quad (14.11)$$

のようにペアの片側だけに作用するブラケットだけを考えておけば良くなるからである。

* * *

カテゴリは関数の合成演算子をオーバーライドする仕組みであった。いま欲しいのは、関数をオーバーライドし、かつブラケットを生成できる仕組みである。

そのような型全体が所属する型クラスを**アロー型クラス**と呼ぶ。アロー型クラスは **Arrow** と表記する。アロー型クラスのインスタンスを **a** で表すでしょう。もちろん **a** は型コンストラクタである。

a 型コンストラクタによって作られる値を**アロー**と呼ぶ。アローは関数から作られるものとする。関数 f が $\mathbf{b} \rightarrow \mathbf{c}$ という型を持つとき、関数 f から作られたアローを \vec{f} と書き、アロー \vec{f} は $\mathbf{b} \xrightarrow{a} \mathbf{c}$ という型を持つとする。

型 $\mathbf{b} \rightarrow \mathbf{c}$ は $(\rightarrow)\mathbf{bc}$ の中置記法バージョンであった。我々の $\mathbf{b} \xrightarrow{a} \mathbf{c}$ も $(\xrightarrow{a})\mathbf{bc}$ の中置記法バージョンである。ただし (\xrightarrow{a}) は単純に **a** と書いて良い。

* * *

関数 f をアロー \vec{f} にする演算子 (アロー演算子) を用意しよう。これを

$$\vec{f} = \langle\langle f \rangle\rangle \quad (14.12)$$

とする。

アロー \vec{f} からブラケットを作る演算子 `first` ただし

$$\begin{bmatrix} f \\ \text{id} \end{bmatrix} = \text{first } \vec{f} \quad (14.13)$$

を用意する. 左辺のブラケットもまたアローであるとする.

ふたつのアローを合成する演算子を

$$\vec{f} \bullet \vec{g} = \begin{bmatrix} f_{1,1} \cdot f_{2,1} \\ f_{1,2} \cdot f_{2,2} \end{bmatrix} \quad (14.14)$$

と定義する.

最後にアローの `id` 関数を改めて定義しておこう. アロー $\vec{\text{id}}$ は

$$\vec{\text{id}} \bullet \vec{f} = \vec{f} \bullet \vec{\text{id}} = \vec{f} \quad (14.15)$$

とする.

これらの実装を与える. アロー型クラスは次のように定義されている.

$$\begin{aligned} &\text{typeclass}^\dagger \text{ Category } \supset \text{a} \Rightarrow \text{Arrow} \supset \text{a} \\ &\text{where } \begin{cases} \langle\!\langle \rangle\!\rangle :: (\text{b} \rightarrow \text{c}) \rightarrow (\text{b} \xrightarrow{\text{a}} \text{c}) \\ \text{first} :: (\text{b} \xrightarrow{\text{a}} \text{c}) \rightarrow \left(\begin{pmatrix} \text{b} \\ \text{d} \end{pmatrix} \xrightarrow{\text{a}} \begin{pmatrix} \text{c} \\ \text{d} \end{pmatrix} \right) \end{cases} \quad (14.16) \end{aligned}$$

つまり値コンストラクタ $\langle\!\langle f \rangle\!\rangle$ と関数 `first` を実装した新しい型を用意すればよいことになる. いや, アロー型クラスはカテゴリ型クラスから派生しているので, カテゴリ型クラスが備える関数合成 (\bullet) と `id` 関数 (`id`) の実装も必要である.

これらを踏まえて, 新しい型コンストラクタ `SimpleFunc` を考えてみる.

$$\text{newtype SimpleFunc a b} = \text{SimpleFunc} \{ \text{funcall} :: \text{a} \rightarrow \text{b} \} \quad (14.17)$$

型 `SimpleFunc a b` は $\text{a} \rightarrow \text{b}$ 型の変数を唯一のメンバに持つ型で, つまりは 1 引数 1 戻値の関数をラップしただけのものである.

この型コンストラクタ *SimpleFunc* を型クラス *Arrow* のインスタンスにしてみよう.

$$\begin{aligned} &\text{instance}^\dagger \text{ } \textcolor{red}{Arrow} \supset \textcolor{red}{SimpleFunc} \\ &\text{where } \left\{ \begin{array}{l} \langle\!\langle f \rangle\!\rangle = \text{SimpleFunc} \llbracket f \rrbracket \\ \text{first } \text{SimpleFunc} \llbracket f \rrbracket = \text{SimpleFunc} \llbracket \left(\begin{array}{c} \textcolor{blue}{x} \\ \textcolor{blue}{y} \end{array} \right) \mapsto \left(\begin{array}{c} \textcolor{blue}{f x} \\ \textcolor{blue}{y} \end{array} \right) \rrbracket \end{array} \right. \quad (14.18) \end{aligned}$$

これがブラケットの正体である. $\mathbf{TK} \vdash \vdash$

型コンストラクタ *SimpleFunc* はまた型クラス *Arrow* のインスタンスでもなければならぬので,

$$\begin{aligned} &\text{instance}^\dagger \text{ } \textcolor{red}{Category} \supset \textcolor{red}{SimpleFunc} \\ &\text{where } \left\{ \begin{array}{l} \text{id} = \langle\!\langle \text{id} \rangle\!\rangle \\ \text{SimpleFunc} \llbracket f \rrbracket \bullet \text{SimpleFunc} \llbracket g \rrbracket = \text{SimpleFunc} \llbracket f \bullet g \rrbracket \end{array} \right. \quad (14.19) \end{aligned}$$

のように *id* と \bullet の実装を与える.

ユーティリティとして, 関数 *split* と関数 *unsplit* のアローバージョンを用意しておく.

$$\left\| \begin{array}{l} \text{split}' :: \textcolor{red}{Arrow} \supset \textcolor{red}{a} \Rightarrow \textcolor{red}{b} \xrightarrow{\textcolor{red}{a}} (\textcolor{red}{b}, \textcolor{red}{b}) \\ \text{split}' = \langle\!\langle \text{split} \rangle\!\rangle \end{array} \right. \quad (14.20)$$

$$\left\| \begin{array}{l} \text{unsplit}' :: \textcolor{red}{Arrow} \supset \textcolor{red}{a} \Rightarrow (\textcolor{red}{b} \rightarrow \textcolor{red}{c} \rightarrow \textcolor{red}{d}) \rightarrow \left(\left(\begin{array}{c} \textcolor{red}{b} \\ \textcolor{red}{c} \end{array} \right) \xrightarrow{\textcolor{red}{a}} \textcolor{red}{d} \right) \\ \text{unsplit}' = \langle\!\langle \text{unsplit} \rangle\!\rangle \end{array} \right. \quad (14.21)$$

型クラス *Arrow* には関数 *second* のデフォルト実装

$$\text{second} = \langle\!\langle \text{swap} \rangle\!\rangle \bullet \text{first} \text{ where } \text{swap} \left(\begin{array}{c} \textcolor{blue}{x} \\ \textcolor{blue}{y} \end{array} \right) \triangleq \left(\begin{array}{c} \textcolor{blue}{y} \\ \textcolor{blue}{x} \end{array} \right) \quad (14.22)$$

が与えられており, この *second* を使ってブラケットの定義

$$\left[\begin{array}{c} f \\ g \end{array} \right] = \text{second } g \bullet \text{first } f \quad (14.23)$$

が与えられている。Haskell ではこのブラケットを `f***g` と表現する。

もう一つ便利なユーティリティ関数 `liftA2` が定義されており、

$$\text{liftA}_2 \star fg = \langle\langle \text{unsplit} \star \rangle\rangle \bullet \text{second } g \bullet \text{first } f \bullet \langle\langle \text{split} \rangle\rangle \quad (14.24)$$

である。

使ってみる。

$$f, g :: \text{SimpleFunc} [\text{Int Int}] \quad (14.25)$$

$$f = \langle\langle (/2) \rangle\rangle \quad (14.26)$$

$$g = \langle\langle \backslash x \mapsto x * 3 + 1 \rangle\rangle \quad (14.27)$$

$$h = \text{liftA}_2(+)fg \quad (14.28)$$

$$z = (\text{funcall } h)8 \quad (14.29)$$

結果として $z = 29$ を得る。

$$\text{sum}' = \text{SimpleFunc} [\text{sum}] \quad (14.30)$$

$$\text{length}' = \text{SimpleFunc} [\text{length}] \quad (14.31)$$

$$\text{mean}' = \text{liftA}_2(/) \text{sum}' \text{length}' \quad (14.32)$$

$$\text{mean}' = \text{unsplit}'(/) \bullet \text{second } \text{sum}' \bullet \text{first } \text{length}' \bullet \text{split}' \quad (14.33)$$

$$= \text{unsplit}'(/) \bullet \left[\begin{array}{c} \text{length}' \\ \text{sum}' \end{array} \right] \bullet \text{split}' \quad (14.34)$$

$$m = \text{funcall } \text{mean}' x_s \quad (14.35)$$

$$\begin{pmatrix} fx \\ gy \end{pmatrix} = \left(\begin{bmatrix} f \\ \text{id} \end{bmatrix} \bullet \begin{bmatrix} \text{id} \\ g \end{bmatrix} \right) \begin{pmatrix} x \\ y \end{pmatrix} \quad (14.36)$$

—

$$\langle\langle f \rangle\rangle = \vec{f} \quad (14.37)$$

$$\text{first } \vec{f} = \begin{bmatrix} f \\ \text{id} \end{bmatrix} \quad (14.38)$$

$$\vec{f} \bullet \vec{g} = \begin{bmatrix} f \bullet g_1 \\ g \bullet g_2 \end{bmatrix} \text{ where ...} \quad (14.39)$$

—

議論を簡単にするために

$$\begin{pmatrix} fx \\ y \end{pmatrix} = \begin{bmatrix} f \\ \text{id} \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (14.40)$$

と

$$\begin{pmatrix} x \\ gy \end{pmatrix} = \begin{bmatrix} \text{id} \\ g \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (14.41)$$

を考えよう。この二つを合成すると

$$\begin{bmatrix} fx \\ gy \end{bmatrix} = \begin{bmatrix} \text{id} \\ g \end{bmatrix} \bullet \begin{bmatrix} f \\ \text{id} \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (14.42)$$

であるとする。つまり、合成則も一緒に考えておくのである。

合成と言えばカテゴリである。つまりブラケットは、カテゴリのインスタンスでなくてはならない。

—

$$\text{id} = \begin{bmatrix} \text{id} \\ \text{id} \end{bmatrix} \quad (14.43)$$

—
 ブラケットの型をカプラと呼ぶことにしよう.

いま関数 first があり,

$$\begin{bmatrix} f \\ \text{id} \end{bmatrix} = \text{first } f \quad (14.44)$$

であるならば嬉しい.

あとはブラケットとペアの間の関数適用さえ定義できればいい.

いや、ちょっと待った. ブラケットの合成はどうなるのだろう.

$$f \xrightarrow{\text{first}} \begin{bmatrix} f \\ \text{id} \end{bmatrix} \quad (14.45)$$

このブラケットは関数だろうか.

これをアローと呼ぶ.

ならば f もアローであるべきだ.

これまでの関数の型は $\mathbf{a} \rightarrow \mathbf{b}$ であった. これから新しい「関数風」の型 $\mathbf{a} \xrightarrow{c} \mathbf{b}$ を考える.

$$\vec{f} = \langle\langle f \rangle\rangle \quad (14.46)$$

$$\text{first } \vec{f} = \backslash \begin{pmatrix} x \\ z \end{pmatrix} \mapsto \begin{pmatrix} fx \\ z \end{pmatrix} \quad (14.47)$$

$$\begin{bmatrix} f \\ \text{id} \end{bmatrix} = \text{first } \vec{f} \quad (14.48)$$

$$\begin{bmatrix} f \\ g \end{bmatrix} = \text{first } \vec{f} \bullet \text{second } \vec{g} \quad (14.49)$$

—

より抽象度の高い議論を行うと

$$\begin{pmatrix} fx \\ y \end{pmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (14.50)$$

さえあれば良いことがわかる.

ただし \vec{f}, \vec{g} はもはや関数ではなく, $\vec{f} *** \vec{g}$ も関数ではない.

—

$$\begin{aligned} \text{typeclass}^\dagger (\text{Category} \supset c) &\Rightarrow (\text{Arrow} \supset c) \\ \text{where } \left\{ \begin{array}{l} \langle\langle \cdots \rangle\rangle :: (x \rightarrow y) \rightarrow c \ x \ y \\ \text{first} :: c \ x \ y \rightarrow c \begin{pmatrix} x \\ z \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \end{array} \right. \quad (14.51) \end{aligned}$$

$$z = g \bullet fx \quad (14.52)$$

$$z = \text{app} \left(\begin{array}{c} \langle\langle \text{uncurry } g \rangle\rangle \bullet \text{first} \langle\langle f \rangle\rangle \bullet \langle\langle \lambda x \mapsto \begin{pmatrix} x \\ x \end{pmatrix} \rangle\rangle \\ x \end{array} \right) \quad (14.53)$$

—

$$\text{newtype } \text{Circ } a \ b = \text{Circ} [\text{uncircuit} :: a \rightarrow (\text{Circ } a \ b, b)] \quad (14.54)$$

instance *Category* \supset *Circ*

$$\text{where} \left\{ \begin{array}{l} \text{id} = \text{Circ} \llbracket \backslash x \mapsto (\text{id}, x) \rrbracket \\ \text{Circ} \llbracket g \rrbracket \bullet \text{Circ} \llbracket f \rrbracket = \text{Circ} \llbracket \backslash x \mapsto (g' \bullet f', z) \\ \text{where} \left\{ \begin{array}{l} \left(\begin{smallmatrix} f' \\ y \end{smallmatrix} \right) \triangleq f x; \left(\begin{smallmatrix} g' \\ z \end{smallmatrix} \right) \triangleq g y \end{array} \right\} \end{array} \right\} \quad (14.55)$$

instance *Arrow* \supset *Circ*

$$\text{where} \left\{ \begin{array}{l} \langle\langle f \rangle\rangle = \text{Circ} \llbracket \backslash x \mapsto \left(\begin{smallmatrix} \langle\langle f \rangle\rangle \\ f x \end{smallmatrix} \right) \rrbracket \\ \text{first}_{\text{Circ}} \llbracket f \rrbracket = \text{Circ} \llbracket \backslash xy \mapsto \left(\begin{smallmatrix} \text{first } f' \\ (z) \\ y \end{smallmatrix} \right) \text{ where } \left(\begin{smallmatrix} f' \\ z \end{smallmatrix} \right) \triangleq f x \end{array} \right\} \quad (14.56)$$

$$\text{runCircuit} :: \text{Circ } \mathbf{a} \ \mathbf{b} \rightarrow [\mathbf{a}] \rightarrow [\mathbf{b}] \quad (14.57)$$

$$\text{runCircuit } _ [] = [] \quad (14.58)$$

$$\text{runCircuit } c(x : x_s) = x' : \text{runCircuit } c' x_s \quad (14.59)$$

$$\text{where } (c', x') \triangleq \text{uncircuit } c x \quad (14.60)$$

$$\text{accum } a \star = \text{Circ} \llbracket \backslash x \mapsto (\text{accum } a' \star, y) \text{ where } (y', a') \triangleq x \star a \rrbracket \quad (14.61)$$

$$\text{accum}' a \star = \text{accum } a (\backslash xy \mapsto (y', y') \text{ where } y' \triangleq x \star y) \quad (14.62)$$

$$\text{total} :: \text{Circ } \mathbf{a} \ \mathbf{a} \quad (14.63)$$

$$\text{total} = \text{accum}' 0 (+) \quad (14.64)$$

$$\text{mean} :: \textcolor{red}{Circ} \textcolor{red}{a} \textcolor{red}{a} \quad (14.65)$$

$$\text{mean} = (\text{total} \bigvee \langle\langle \text{const } 1 \rangle\rangle \Rightarrow \text{total}) \Rightarrow \langle\langle \text{uncurry}(/) \rangle\rangle \quad (14.66)$$

—

$$\text{newtype } \textcolor{red}{Kleisli} \textcolor{red}{m} \textcolor{red}{a} \textcolor{red}{b} = \text{Kleisli} [\text{runKleisli} :: \textcolor{red}{a} \rightarrow^m \llbracket \textcolor{red}{b} \rrbracket] \quad (14.67)$$

$$\backslash x \mapsto ((\backslash y \mapsto \text{Just} \llbracket y * 2 \rrbracket) \heartsuit \text{Just} \llbracket x * 3 \rrbracket) = (\text{Just} \llbracket \diamond \rrbracket \cdot (*2)) \blacktriangleright (\text{Just} \llbracket \diamond \rrbracket \cdot (*3)) \quad (14.68)$$

$$\begin{aligned} \backslash x \mapsto ((\backslash y \mapsto \text{Just} \llbracket y * x \rrbracket) \heartsuit \text{Just} \llbracket x * 2 \rrbracket) \\ = (\text{Just} \llbracket \diamond \rrbracket \cdot \text{uncurry}(*)) \blacktriangleright (\text{Just} \llbracket \diamond \rrbracket \cdot ((*2), \text{id})) \end{aligned} \quad (14.69)$$

—

$$(f \bigvee g) \begin{pmatrix} \textcolor{blue}{x} \\ \textcolor{blue}{y} \end{pmatrix} = \begin{pmatrix} \textcolor{blue}{fx} \\ \textcolor{blue}{gy} \end{pmatrix} \quad (14.70)$$

$$(f \bigwedge g) x = \begin{pmatrix} \textcolor{blue}{fx} \\ \textcolor{blue}{gx} \end{pmatrix} \quad (14.71)$$

—

第 15 章

プログラム

A...

15.1 文字セットとコメント

Haskell コンパイラを含む多くのコンパイラが Unicode 文字セットに対応しているものの、従来からの習慣や英語圏での使いやすさを考慮してか、ASCII 文字セットだけでプログラムを書けるようにしているし、またそれを推奨している。

Haskell プログラムもまた、文字定数を除いては ASCII 文字セットの範囲で書くことが普通である。そこで我々もその習慣に従うことにしよう。例えば円周率を代入する変数を π と書きたいところだが、我々は `pi` と書く。

数学記号のほとんども、ASCII 文字セットの中から記号を組み合わせるか、さもなくば言葉で表現する。例えば Haskell では \in の代わりに `<-` を使うし、 \neg の代わりに `not` を使う。

また計算機科学者たちの絶えざる努力にもかかわらず、プログラム中の文字の装飾はこれまでほとんど受け入れられていない。我々は本書で `f`, *f*, *f*, *f*, *f*, *f*, *f* を使い分けてきたが、Haskell プログラム中では全て `f` と書く。

以上のような制約にもかかわらず、Haskell プログラムと我々が見えきた

「カーリー風の」数学記法は本質的に差がない．本書を読み進めてきた読者なら，Haskell プログラムを読むのに苦労はいらないだろう．

Haskell では `--` から行の終わりまでがコメントとして扱われる．また `{-` で始まり `-}` で終わる文字列もコメントとして扱われる．

* * *

習慣的に Haskell プログラムのファイルには拡張子 `.hs` を付ける．

15.2 main 関数と一般の関数定義

Python インタプリタはプログラムを頭から実行していくので，`main` 関数は書かなくてもよいが，アプリケーションプログラマにとっての一番の関心事は `main` 関数（Haskell では `main` アクション）の書き方だろう．iOS アプリケーション開発のように，基本的には `main` 関数をプログラマが触れないというスタイルもあるが，それでもデバッグの時には `main` 関数から辿ることになるので，`main` 関数のありかを知っておくことはいつでも重要だ．

Haskell コンパイラも `main` アクションをアプリケーションプログラムのエントリポイントとして認識する．というよりも，Haskell プログラムとは `main` という一つのアクションである．`main` を含む一般の関数やアクションはこれまで通り

```
main = ... (15.1)
```

のように関数名のあとに等号 (`=`) を置いて定義する．

C で `main` 関数の型が OS の都合で `int main(void)` または `int main(int, const char *const *)` と決められているように，Haskell でも `main` アクションの型はあらかじめ決められている．その型は ¹⁰`[Int]` である．

UNIX および UNIX に影響を受けた OS では，プログラムは終了時に整数値を OS へ返すことになっている．プログラムが 0 を返せば，そのプロ

グラムは正常終了したとみなされる。例えば UNIX シェル (sh や csh のこと) で,

```
$ program1 && program2
```

としたとき, program1 の戻り値が 0 のときに限って program2 が実行される。

何もせずに OS に 0 を返すプログラムすなわち

$$\left\| \begin{array}{l} \text{main} :: \text{IO} \llbracket \text{Int} \rrbracket \\ \text{main} = \llbracket 0 \rrbracket \end{array} \right\| \quad (15.2)$$

を Haskell で書くと,

```
-- do-nothing.hs
main :: IO Int
main = pure 0
```

Haskell

または

```
-- do-nothing.hs
main :: IO Int
main = return 0
```

Haskell

のようになる。ここに return はモナドのピュア演算子の別名で、特別に **ユニット演算子** と呼ぶ。^{*1}

このプログラムは C の

^{*1} GHC v7.8 以前はモナドにピュア演算子が定義されず、モナド独自のユニット演算子が定義されていた。

C

```
/* do-nothing.c */
int main(void) {
    return 0;
}
```

と等しい。

15.3 プログラミングの本質

副作用のないプログラムはひとつの関数で書ける。その関数を `main` と呼ぶことにすると、この `main` 関数を

$$\text{main} = f_n \cdot f_{n-1} \cdot \cdots \cdot f \quad (15.3)$$

と部分関数に分解することがプログラマの能力である。副作用のないプログラムとは、コマンドラインから引数 x を受け取り、なんらかの処理を行い、OS に終了値を返すだけのプログラムである。このようなプログラムは普通役に立たないが、議論が簡単になるので少し見てみよう。

副作用のない関数を合成するのはわけのないことだ。もし Python ならば

Python

```
y = f2(f1(x))
```

のように関数を入れ子にしても良いし、読みづらければ途中経過を一時変数にして

Python

```
y1 = f1(x)
y2 = f2(y1)
```

としてもよい。副作用のない関数の場合、これが関数合成の規則である。

プログラムが副作用を持つ場合は、プログラム自身が IO モナドであるため、関数へと分解できないのであった。副作用を持つプログラムはアクションであり、そのアクションを `main` アクションと名付けると、その `main` ア

クシオンを

$$\text{main} = \alpha_n \heartsuit \alpha_{n-1} \heartsuit \cdots \heartsuit \alpha_0 \quad (15.4)$$

と部分アクションに分解することがプログラマの能力となる.

* * *

ここで、プログラマの「サバイバルキット」を用意しておこう. 次のプログラムで定数 x と関数 f の部分を埋めれば、関数適用の結果つまり $f\ x$ の値を画面に出力する.

Haskell

```
-- survivalkit.hs
x :: Double
x = {- Value -}
f :: Double -> Double
f = {- Function -}
main :: IO Int
main = print (f x) >> pure 0
```

ファイル名を `survivalkit.hs` とすると、コンパイルと実行は次のようにする.

```
$ ghc survivalkit.hs
$ ./sample
```

プログラム `survivalkit.hs` を数式で書くと

$$\begin{array}{l} \parallel x :: \text{Double} \\ \parallel x = \dots \end{array} \quad (15.5)$$

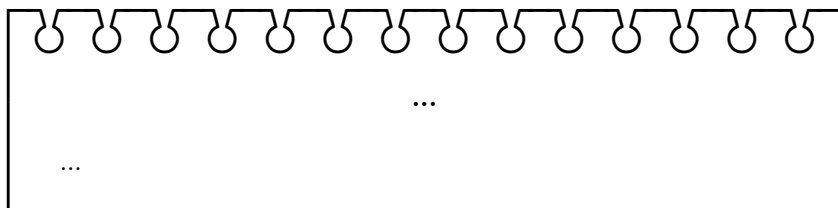
$$\begin{array}{l} \parallel f :: \text{Double} \rightarrow \text{Double} \\ \parallel f = \dots \end{array} \quad (15.6)$$

$$\begin{array}{l} \parallel \text{main} :: {}^{\text{IO}} \llbracket \text{Double} \rrbracket \\ \parallel \text{main} = \text{print}(fx) \rightarrow \llbracket 0 \rrbracket \end{array} \quad (15.7)$$

である. この式は $\text{print}(fx)$ がいかなる値を返そうとも、アクション main の値は 0 になることを示している.

15.4 余談：インタプリタ*

15.5 この章のまとめ*



~~TK: TO DO~~

- state monad
- fix
- category
- kleisli class
- arrow
- continuation
- 文字セットとコメント
- 代数的構造
- ラムダ

第Ⅱ部

執筆用×毛*

第 16 章

演算*

演算の例を挙げる．

16.1 名前と予約語*

表 16.1 および表 16.2 に Haskell で予約されている記号と名前を掲げる．これらの記号，名前は演算子，関数，定数として使うことが出来ない．

もちろんこれら以外に `+` や `map` など慣例的に使われている語の再定義は避けるべきである．特に `Prelude` モジュールは特段の事情がない限り必ず読み込まれるモジュールなので，`Prelude` で定義される記号と名前の再定義は避けるべきであろう．`Prelude` で定義される記号と名前は膨大な数になるので，リファレンスを参考にしてもらいたい．

Haskell で演算子に使える記号は

`! @ # $ % ^ & * - + = . \ | / < : > ? ~`

である．ただし `:` ではじまる記号は予約されている．

表 16.1 Haskell の予約済み演算子と記号

--	行コメント
{--}	コメント
'	文字リテラル
"	文字列リテラル
`	関数の中置
()	括弧
{ }	レコード構文 ({...}), ブロック
;	ステートメントセパレータ
\	ラムダ (\), 行分割
:	結合演算子
::	型の宣言 (::)
..	等差数列
=	定義 (=)
=>	インスタンス定義 (⇒)
->	関数型コンストラクタ (→), ラムダ式の矢印 (→), case 式の矢印 (→)
<-	リスト内包表記 (∈), do 記法中の代入 (←)
@	as パタン (@)
	ガード (), リスト内包 (), データ型の和演算 (∇)
,	リストの値の区切り, リスト内包表記の区切り, タプルの値の区切り
!	正格評価
~	XYZ

表 16.2 Haskell の予約語

分岐 (1)	case, of
分岐 (2)	if, then, else
型定義	class, data, deriving, instance, newtype, type
do 記法	do
局所変数 (1)	let, in
局所変数 (2)	where
演算子定義	infix, infixl, infixr
モジュール関係	foreign, import, module
ワイルドカード引数	_

16.2 2 次方程式の解*

2 次方程式の解とは、定数 a, b, c が既知の式

$$a * x^2 + b * x + c = 0 \quad (16.1)$$

において定数 x が取り得る値のことである。なお x^2 は Haskell では $x**2$ と書く。

2 次方程式の解法は知られており、

$$(x_0, x_1) = (q/a, c/q) \quad \text{where } q \triangleq \frac{(b + (\text{sgn } b) * \sqrt{\text{sq } b - 4 * a * c})}{2} \quad (16.2)$$

である。ここに関数 sq は

$$\text{sq } x = x^2 \quad (16.3)$$

である。わざわざ関数にしたのは、単純な `x**2` よりも高速な実装

$$\left\| \begin{array}{l} \text{sq} :: \text{Real} \supset \mathbf{a} \Rightarrow \mathbf{a} \rightarrow \mathbf{a} \\ \text{sq } x = x * x \end{array} \right. \quad (16.4)$$

を後で与えるためである。平方根 \sqrt{x} は Haskell では `sqrt x` と書く。

また関数 `sgn` は

$$\left\| \begin{array}{l} \text{sgn} :: \text{Real} \supset \mathbf{a} \Rightarrow \mathbf{a} \rightarrow \mathbf{a} \\ \text{sgn } x \mid_{x < 0} = -1 \\ \qquad \qquad \mid_{\text{otherwise}} = 1 \end{array} \right. \quad (16.5)$$

である。

最後にタプル (x_0, x_1) を画面に表示すれば完了なので

$$\left\| \begin{array}{l} \text{main} :: \text{IO} \llbracket \text{Int} \rrbracket \\ \text{main} = \text{print}(x_0, x_1) \rightarrow \llbracket 0 \rrbracket \end{array} \right. \quad (16.6)$$

とする。

これをそのまま Haskell で実装すれば、2 次方程式の解が求まる。

Haskell

```
-- quadratic.hs
a = {- ... -}
b = {- ... -}
c = {- ... -}
sgn :: Real a => a -> a
sgn x | x < 0      = -1
      | otherwise = 1
sq :: Real a => a -> a
sq x = x*x
(x0, x1) = (q/a, c/q) where
  q = (b+sgn(b)*sqrt(sq b - 4*a*c))/2.0
main :: IO Int
main = print (x0, x1) >> pure 0
```

なお、2 次方程式のよく知られた解法、すなわち

$$(x_0, x_1) = \left(\frac{-b+d}{2*a}, \frac{-b-d}{2*a} \right) \text{ where } d \triangleq \sqrt{b^2 - 4*a*c} \quad (16.7)$$

は定数 a または c の値が小さい場合には計算誤差が大きくなるため、推奨されない。

16.3 複素数*

2 次方程式の定数 a, b, c が複素数だとして、それぞれの実部を添え字 r で、虚部を添え字 i で区別することになると

$$a = a_r + a_i \quad (16.8)$$

$$b = b_r + b_i \quad (16.9)$$

$$c = c_r + c_i \quad (16.10)$$

となる。ここに $+$ は複素数を合成する演算子で、第 1 引数（左引数）が実部、第 2 引数（右引数）が虚部からなる複素数を合成するものとする。Haskell では

```
import Data.Complex
a :: Complex Double
a = ar :+ ai
```

Haskell

のように $+$ 演算子を使って複素数をコンストラクトする。

このように Haskell では複素数を

$$^c[a] = \text{Complex } a \quad (16.11)$$

の型パラメタ a に **Float** 型または **Double** 型を当てはめて構築した型を用いる。**Float** 型および **Double** 型は **RealFloat** 型クラスに属する。

複素数の場合の 2 次方程式の解は、実数の場合とほぼ同じで

$$(x_0, x_1) = (q/a, c/q) \text{ where } \begin{cases} q \triangleq (b + \text{sgn}' abc * r) / 2 \\ r \triangleq \text{sqrt}' (\text{sq}' b - 4 * a * c) \end{cases} \quad (16.12)$$

である。実数の場合との違いは **sq** 関数と **sgn** 関数をそれぞれ **sq'** 関数と

sgn' 関数に置き換えたことで,

$$\left\| \begin{array}{l} \text{sq}' :: \text{RealFloat} \supset \mathbf{a} \Rightarrow \mathbf{a} \rightarrow {}^C\llbracket \mathbf{a} \rrbracket \rightarrow {}^C\llbracket \mathbf{a} \rrbracket \\ \text{sq}' x = x * x \end{array} \right. \quad (16.13)$$

および

$$\left\| \begin{array}{l} \text{sgn}' :: \text{RealFloat} \supset \mathbf{a} \\ \Rightarrow {}^C\llbracket \mathbf{a} \rrbracket \rightarrow {}^C\llbracket \mathbf{a} \rrbracket \rightarrow {}^C\llbracket \mathbf{a} \rrbracket \rightarrow \mathbf{a} \\ \text{sgn}' abc \mid \Re((\text{conjugate } b) * \text{sq}' (b - 4 * a * c)) < 0 = -1 \\ \mid \text{otherwise} = 1 \end{array} \right. \quad (16.14)$$

である.

この sgn' 関数の中で複素数の平方根および複素数の実数倍を計算する必要がある. そこで, 複素数の平方根を

$$\left\| \begin{array}{l} \text{sqrt}' :: \text{RealFloat} \supset \mathbf{a} \Rightarrow \mathbf{a} \rightarrow \mathbf{a} \\ \text{sqrt}'(x \dot{+} y) = \frac{\sqrt{2}}{2} * \left(\sqrt{d + x} \dot{+} (\text{sgn } y) * \sqrt{d - x} \right) \\ \text{where } d \triangleq \sqrt{\text{sq } x + \text{sq } y} \end{array} \right. \quad (16.15)$$

と定義し, 複素数の実数倍は演算子 $*$ で表すとして

$$\left\| \begin{array}{l} (\dot{*}) :: \text{RealFloat} \supset \mathbf{a} \Rightarrow \mathbf{a} \rightarrow {}^C\llbracket \mathbf{a} \rrbracket \rightarrow {}^C\llbracket \mathbf{a} \rrbracket \\ (\dot{*}) a(x \dot{+} y) = a * x \dot{+} a * y \end{array} \right. \quad (16.16)$$

とした. (演算子 $*$ は $*$. と書くことにする.)

Haskell は複素数の実数倍のための演算子を標準では用意していないため, このように独自の演算子を定義するか, あるいは複素数掛ける複素数の形にして演算を行うかのいずれかが必要である.

なお, `conjugate` は共役複素数を求める関数, \Re は実部を求める関数である. \Re は Haskell では `realPart` と書く.

まとめると次のようになる.

Haskell

```

-- complex.hs
import Data.Complex

a, b, c :: Complex Double
a = {- ... -} :+ {- ... -}
b = {- ... -} :+ {- ... -}
c = {- ... -} :+ {- ... -}

sq :: Real a => a -> a
sq x = x*x

sq' :: RealFloat a => Complex a -> Complex a
sq' x = x*x

sgn :: Real a => a -> a
sgn x | x < 0      = (-1)
      | otherwise = 1

sgn' :: RealFloat a =>
  Complex a -> Complex a -> Complex a -> Complex a
sgn' a b c
  | realPart((conjugate b)*sqrt'(sq' b-4*(a*c)))<0 = (-1):+0
  | otherwise                                     = 1:+0

sqrt' :: RealFloat a => Complex a -> Complex a
sqrt' (x:+y) = (sqrt 2)/2*((sqrt(d+x)):+sgn(y)*(sqrt(d-x)))
  where d = sqrt(sq x + sq y)

(*.) :: Real a => a -> Complex a -> Complex a
(*.) a (x:+y) = a*x:+a*y

(x0, x1) = (q/a, c/q) where
  q = (b + sgn' a b c * r) / 2
  r = sqrt'(sq' b - 4*(a*c))

main :: IO Int
main = print a >> pure 0

```

16.4 余談：正格評価*

第 17 章

より複雑な演算

17.1 数学関数の演算*

$y = fx$ ただし

$$\begin{aligned} fx|_{x \equiv 0} &= 1 \\ |_{\text{otherwise}} &= (\sin x)/x \end{aligned} \quad (17.1)$$

とする. ここで $0 \leq x \leq \pi$ の範囲で y の値を求めたいとしよう. x の範囲を $n+1$ 分割するとする.

まず n に具体的な型と値を与えておこう. これは次のようにする.

$$\begin{aligned} & n :: \text{Int} \\ & n = 100 \end{aligned} \quad (17.2)$$

次に, 関数 f を定義しておこう.

$$\begin{aligned} & f :: \text{Double} \rightarrow \text{Double} \\ & fx|_{x \equiv 0} = 1 \\ & |_{\text{otherwise}} = (\sin x)/x \end{aligned} \quad (17.3)$$

関数 f に与える引数 x のリストを定義する.

$$\begin{aligned} & x_s :: [\text{Double}] \\ & x_s = [i/n * \pi \mid i \in [0 \dots n]] \end{aligned}$$

実はこのままではまずい．変数 i も変数 n も **Int** 型なので，割り算が出来ないのだ．そこで型変換のための関数 $\triangle_{\text{Integral}}$ を使おう．関数 $\triangle_{\text{Integral}}$ は **Integral** 型クラスの型の変数を，任意の型へと変換する．関数 $\triangle_{\text{Integral}}$ を使って書き直すと

$$\begin{array}{l} \| x_s :: [\text{Double}] \\ \| x_s = [(\triangle_{\text{Integral}} i) / (\triangle_{\text{Integral}} n) * \pi \mid i \in [0 \dots n]] \end{array} \quad (17.4)$$

となる．この結果，リスト x_s は

$$x_s = [0, (1/n) * \pi, (2/n) * \pi, \dots, \pi] \quad (17.5)$$

となる．

最後に，関数 f をリスト x_s に適用して結果を得る．

$$\begin{array}{l} \| y :: [\text{Double}] \\ \| y = f \odot x_s \end{array} \quad (17.6)$$

結果 y_s はプログラム

$$\text{main} = \text{print } y \rightarrow \llbracket 0 \rrbracket \quad (17.7)$$

によって出力できる．

これをそのまま Haskell プログラムにすると次のようになる．

```
-- sample.hs
n :: Int
n = 100
f :: Double -> Double
f x | x == 0    = 1
    | otherwise = (sin x)/x
xs :: [Double]
xs = [(fromIntegral i) / (fromIntegral n)
      * pi | i <- [0..n]]
ys :: [Double]
ys = f `map` xs
main = print ys >> pure 0
```

Haskell

表 17.1 Prelude のデータ型

名前	型名
Bool	論理型 (Bool)
Char	文字型 (Char)
Int	整数型 (Int)
Integer	整数型 (Integer)
Float	単精度浮動小数点型 (Float)
Double	倍精度浮動小数点型 (Double)
String	[Char] の型シノニム

表 17.2 Prelude の多相型

名前	型名
[a]	リスト ([a])
Maybe a	Maybe (Maybe [a])
Either a b	Either (Either [a b])
((->)r)a	関数 (((->) r) [a]r)

なお、円周率 π はそのまま定数 `pi` として Prelude で定義されている。

$$(f \odot g)t = \sum_x f x * g(t - x) \quad (17.8)$$

$$\begin{aligned}
yt &= \sum (\backslash x \mapsto \text{fst } x * \text{snd } x) \odot h_s \\
&\text{where} \\
h_s &\triangleq \text{zip } f_s g_s \\
f_s &\triangleq [f x \mid x \in [0, 1 \cdots n]] \\
g_s &\triangleq [g(t - x) \mid x \in [0, 1 \cdots n]]
\end{aligned} \tag{17.9}$$

効率悪い.

—

$$(\odot) :: (\text{Num} \supset \mathbf{a}) \Rightarrow (\mathbf{a} \rightarrow \mathbf{a}) \rightarrow (\mathbf{a} \rightarrow \mathbf{a}) \rightarrow \mathbf{a} \rightarrow \mathbf{a} \tag{17.10}$$

$$f \odot g = \backslash t \mapsto \sum f x * g(t - x) \tag{17.11}$$

$$yt = (f \odot g)t$$

$$yt = \sum_x f x * g(t - x)$$

$$y = \sum_x f x * g(\diamond - x)$$

17.2 行列演算*

TK: Data.Vector

17.3 余談：アンボックス化

~~TK: Unboxed~~

—

$$z = !x + !y \tag{17.12}$$

Haskell

```
import Data.Complex
z :: Complex Double
z = !x :+ !y
```


第 18 章

IO*

18.1 書き出し*

18.2 読み込み*

18.3 IO モナド*

18.4 余談：コマンドライン引数*

コマンドライン引数はアクション `getArgs` で読み出すことができる。例えば

```
-- args.hs
import System.Environment (getArgs)
main :: IO Int
main = do
  args <- getArgs
  {- do something -}
  return 0
```

Haskell

とすると，リスト `args` にコマンドライン引数が渡される．アクション `getArgs` の戻り値の型は `[String]` である．

$$\textit{main} = (\cdots \text{where } a_s = \textit{getArgs}) \rightarrow \llbracket 0 \rrbracket \quad (18.1)$$

18.5 この章のまとめ*

第 19 章

モジュール*

第 20 章

代数的構造

この章では「代数的構造」を見ていくことにする。代数的構造とは、四則演算のような数に関する基本的な性質を抽象化していくことで、数の背後にある基本的なメカニズムを抽出したものである。代数的構造はあらゆるプログラミング言語に明示的、あるいは非明示的に見られる要素である。

20.1 数

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的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 \mathbb{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 $x, y, z \in \mathbb{R}$ と書くが、本書ではこれまで通り

$$x, y, z :: \mathbb{R} \tag{20.1}$$

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 $x, y, z :: \mathbb{R}$ とする．

実数の性質 1. 加法の全域性 任意の x と任意の y の加法（足し算）の結果すなわち和 $x + y$ は \mathbb{R} の元すなわち実数である．演算の結果が同じ集合の元になることを全域性と呼ぶ．

実数の性質 2. 加法の結合性 任意の x, y, z について

$$(x + y) + z = x + (y + z) \quad (20.2)$$

である．これを加法の結合性（結合律）と呼ぶ．

実数の性質 3. 零元（加法単位元）の存在 特別な実数 $0 :: \mathbb{R}$ があり

$$0 + x = x + 0 = x \quad (20.3)$$

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．単位元が存在することを単位律と呼ぶこともある．

実数の性質 4. 負元（加法逆元）の存在 任意の x に対して $-x :: \mathbb{R}$ があり

$$-x + x = 0 \quad (20.4)$$

である．この $-x$ は x の加法の逆元である．負元または加法逆元と呼ぶこともある．逆元が存在することを消約律と呼ぶこともある．

実数の性質 5. 加法的可換性 任意の x, y について

$$x + y = y + x \quad (20.5)$$

である．このことを加法の可換性（可換律）と呼ぶ．

実数の性質 6. 乗法 任意の x と任意の y の乗法（掛け算）の結果すなわち積 $x * y$ は \mathbb{R} の元すなわち実数である．

実数の性質 7. 乗法の結合性 任意の x, y, z について

$$(x * y) * z = x * (y * z) \quad (20.6)$$

である．

実数の性質 8. 単位元の存在 特別な実数 $1 :: \mathbb{R}$ があり

$$1 * x = x * 1 = x \quad (20.7)$$

である. この 1 を乗法の単位元または**乗法単位元**と呼ぶ.

実数の性質 9. 逆元の存在 任意の x に対して $x^{-1} :: \mathbb{R}$ があり

$$x^{-1} * x = 1 \quad (20.8)$$

である. この x^{-1} は x の乗法の逆元である. **乗法逆元**と呼ぶこともある. ただし性質 11 で述べる通り, 加法単位元については逆元がなくとも良い.

実数の性質 10. 乗法の可換性 任意の x, y について

$$x * y = y * x \quad (20.9)$$

である. このことを乗法の可換性と呼ぶ.

実数の性質 11. 加法単位元の乗法逆元 加法単位元に対する乗法の逆元は存在しなくとも良い. (つまり 0^{-1} のことは考えなくて良い.)

実数の性質 12. 分配律 加法と乗法が混在する場合

$$(x + y) * z = (x * z) + (y * z) \quad (20.10)$$

と乗法を**分配**する.

以上が実数の代数的性質の全てである. 我々がよく使う引き算, 割り算は数学上はシンタックスシュガーである.

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**と呼ぶ.

体の要素は, 集合 \mathbb{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 $(\mathbb{K}, +, 0, -, *, 1, ^{-1})$ と表現する.

体の性質から言えることを一つ紹介しよう．これから

$$z \uparrow n = \underbrace{z * \cdots * z}_n \quad (20.11)$$

なる二項演算子 \uparrow (クヌースの矢印) を使う．ここに z を体の元, n を自然数とした．さて $z \uparrow 2$ は

$$z \uparrow 2 = z * z \quad (20.12)$$

であるから, いま $z = x + y$ とすると

$$z \uparrow 2 = z * z \quad (20.13)$$

$$= z * (x + y) \quad (20.14)$$

$$= z * x + z * y \text{ —分配律} \quad (20.15)$$

$$= (x + y) * x + (x + y) * y \quad (20.16)$$

$$= x * x + y * x + x * y + y * y \text{ —分配律} \quad (20.17)$$

$$= x \uparrow 2 + x * y + y * x + y \uparrow 2 \quad (20.18)$$

となり

$$(x + y) \uparrow 2 = x \uparrow 2 + x * y + y * x + y \uparrow 2 \quad (20.19)$$

を得る．式 (20.19) は体の性質だけを使って導いた関係なので, 実数だけでなく有理数や複素数にもそのまま使える．実際には式 (20.19) は体の性質のうち分配律だけを使っているので, 体以外にも応用が利く式でもある.*¹

20.2 群

体の性質を若干緩めたい場合がある．さもないければ, 整数, 正方行列, クォータニオン (四元数), 論理値, ベクトル, ベクトルの変換, 集合から集合への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．

*¹ Knuth の矢印 (\uparrow) は Haskell では演算子[^]として提供されている．

表 20.1 代表的な代数的構造の性質 (1)

代数的構造	+	+ の単位元	+ の逆元	*	* の単位元	* の逆元
体	可換	あり	あり	可換	あり	あり
環	可換	あり	あり	非可換	あり	なし

例えば整数の掛け算の逆元は（単位元の逆元を除いて）整数の中には存在しないし、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。

だいたいどの辺まで制約を緩めたものを数の仲間に入れるかというのは見解の分かれるところでもあるが、体から性質 9（乗法の逆元）、性質 10（乗法の可換性）、性質 11（加法単位元の乗法逆元）を取り除いたものを環と呼び、環の性質を持つものを数の仲間に入れることが一般的である。環の性質を持つものは、体である実数、有理数、複素数に加えて、整数、正方行列、クォータニオン、論理値などがある。

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ほうがさらなる応用が利きそうである。体には二つの二項演算子 $+$ と $*$ が登場した。その片方のみ注目してみたらどうなるだろう。それがこの節で取り上げる群である。

形式的に体と環の性質を並べたものが表 20.1 である。これを見ると、各演算子について「可換・単位元あり・逆元あり」の組み合わせが二つペアになったもの（体）か、「可換・単位元あり・逆元あり」の組み合わせと「非可換・単位元あり・逆元なし」の組み合わせがペアになったもの（環）があることがわかる。

いま集合 \mathbf{G} があり、 $x, y, z \in \mathbf{G}$ であるとし、二項演算子を \star と書くことにして、体の性質の前半分を書き下してみよう。

性質 1. 任意の x と任意の y の演算の結果 $x \star y$ は \mathbf{G} の元である。

性質 2. 任意の x, y, z について

$$(x \star y) \star z = x \star (y \star z) \quad (20.20)$$

である.

性質 3. 特別な元 $\emptyset :: \mathbf{G}$ があり

$$\emptyset \star x = x \star \emptyset = x \quad (20.21)$$

である.

性質 4. 任意の x に対して $\ominus x :: \mathbf{G}$ があり

$$\ominus x \star x = \emptyset \quad (20.22)$$

である.

性質 5. 任意の x, y について

$$x \star y = y \star x \quad (20.23)$$

である.

このような性質が満たされる時, 組み合わせ $(\mathbf{G}, \star, \emptyset, \ominus)$ を可換群または加群と呼ぶ. この可換群が最初の構成要素「可換・単位元あり・逆元あり」の正体である.

例えば $(\mathbb{R}, +, 0, -)$ は可換群である. 整数全体の集合を \mathbb{Z} とすると $(\mathbb{Z}, +, 0, -)$ も可換群である. また, 集合 \mathbb{R} から 0 だけを取り除いた集合を $\mathbb{R} \setminus 0$ とするとき $(\mathbb{R} \setminus 0, *, 1, ^{-1})$ も可換群である.

可換群は代表的な代数的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 t とする二次元の回転変換を R_t として, 回転変換 R_t すべてからなる集合 \mathbf{R} を考えてみよう. 回転の合成を \cdot で表すとする

$$R_{t_1} \cdot R_{t_2} = R_{(t_1+t_2)} \quad (20.24)$$

であるから、回転を合成した結果も回転である。また式 (20.24) から

$$R_{t_1} \cdot (R_{t_2} \cdot R_{t_3}) = (R_{t_1} \cdot R_{t_2}) \cdot R_{t_3} \quad (20.25)$$

であるから、回転変換は結合性も満たしている。

次に回転変換に単位元があるかどうか調べてみよう。回転しない変換は恒等変換とも言い、しばしば I で表す。何もしない回転変換は 0 度の回転であるから $I = R_0$ である。このとき式 (20.24) から

$$I \cdot R_t = R_t \cdot I = R_t \quad (20.26)$$

であるから、 I 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。 t 回転の逆は明らかに $-t$ であるから

$$R_{-t} \cdot R_t = R_t \cdot R_{-t} = I \quad (20.27)$$

が成り立つ。そこで

$$R_t^{-1} = R_{-t} \quad (20.28)$$

として R_t の逆元 R_t^{-1} を定義することができる。

このように、組み合わせ $(\mathbb{R}, \cdot, I, ^{-1})$ も可換群である。(回転 R_t 全体の集合 \mathbf{R} が群を形成することは、パラメタ t が所属する実数全体の集合 \mathbb{R} が群を形成することに大いに頼っている。この部分を詳細に調べるとリー群という美しい代数的構造が見つかる。)

回転されるものをベクトルと呼ぶ。ベクトルや回転変換の実装方法はいくつかあり、例えば第 1 座標値を u 、第 2 座標値を v としたときにベクトル \vec{p} を

$$\vec{p} = \begin{bmatrix} u \\ v \end{bmatrix} \quad (20.29)$$

と表すことにしよう。矢印は変数 p がベクトルであることを忘れないようにするための飾りである。このとき、回転変換は

$$R_t = \begin{bmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{bmatrix} \quad (20.30)$$

と行列で表すことになり、回転後のベクトル \vec{p}' は

$$\vec{p}' = R_t * \vec{p} \quad (20.31)$$

となる．ここに演算子 $*$ は行列の積である．また、この場合変換の合成 \cdot は行列積 $*$ となる．

他にもベクトルを複素数で表現する方法もある．いま \vec{p} を

$$\vec{p} = u + Iv \quad (20.32)$$

と表して、回転変換 R_t を

$$R_t = \cos t + I \sin t \quad (20.33)$$

とすると、回転後の \vec{p}' はやはり

$$\vec{p}' = R_t * \vec{p} \quad (20.34)$$

と書ける．ここに演算子 $*$ は複素数の積である．また、この場合変換の合成 \cdot も複素数積 $*$ となる．

任意次元のベクトル全体からなる集合を \mathbf{V} として、零ベクトルを $\vec{0}$ で表すことにしよう．ここでも矢印はベクトルであることを忘れないようにするための飾りである．ベクトル同士の加算を二項演算子 $+$ で表し、向きを反転させた逆ベクトル作る演算子を $-$ とすると、組み合わせ $(\mathbf{V}, +, \vec{0}, -)$ もまた可換群である．

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 $x \star y \neq y \star x$ 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．

もう一度組み合わせ $(\mathbf{G}, \star, O, \ominus)$ が群である条件を少し緩め、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．これが構成要素「非可換・単位元あり・逆元なし」の正体である．

表 20.2 代表的な代数的構造の性質 (2)

代数的構造	★	★ の単位元	★ の逆元
可換群	可換	あり	あり
群	非可換	あり	あり
単位的半群	非可換	あり	なし
半群	非可換	なし	なし

単位的半群の性質は次の三つである．ただし x, y, z が集合 \mathbf{M} の元であるとする．

単位的半群の性質 1. 任意の x と任意の y の演算の結果 $x \star y$ は \mathbf{M} の元である．

単位的半群の性質 2. 結合性 任意の x, y, z について

$$(x \star y) \star z = x \star (y \star z) \quad (20.35)$$

である．

単位的半群の性質 3. 単位元の存在 特別な元 $\emptyset :: \mathbf{M}$ があり

$$\emptyset \star x = x \star \emptyset = x \quad (20.36)$$

である．

このとき，組み合わせ $(\mathbf{M}, \star, \emptyset)$ が単位的半群である．可換群とこの単位的半群を組み合わせたのが環，可換群二つを組み合わせたのが体であった．

なお，これまで単位元の定義として

$$\emptyset \star x = x \star \emptyset = x \quad (20.37)$$

を掲げているが，厳密には単位元は

$$\emptyset_{\text{left}} \star x = x \star \emptyset_{\text{right}} = x \quad (20.38)$$

のように、左単位元と右単位元を区別しても良い。

単位的半群の性質からさらに性質 3 を消したものを半群と呼ぶ。可換群，群，単位的半群，半群を一覧にしたものを表 20.2 に掲げる。

20.3 圏

これまでは集合の元同士に対する二項演算を考えてきた。集合 \mathbf{M} が単位的半群であるとき，集合 \mathbf{M} の元 $x, y :: \mathbf{M}$ に対して $x \star y :: \mathbf{M}$ であった。見方を変えると，演算子 \star とは集合 \mathbf{M} の元 2 個から出発して，集合 \mathbf{M} の元 1 個へとジャンプさせる写像であると言える。これを

$$\star :: (\mathbf{M} \times \mathbf{M}) \rightarrow \mathbf{M} \quad (20.39)$$

と書く。ここに $\mathbf{X} \times \mathbf{Y}$ は集合 \mathbf{X} と集合 \mathbf{Y} の直積集合である。直積集合と元の集合はもはや別な集合であることに注意しよう。

写像は $\mathbf{M} \times \mathbf{M} \rightarrow \mathbf{M}$ に限ったもの出はなく，集合 \mathbf{M} から集合 \mathbf{M} への写像 \star ただし

$$\star :: \mathbf{M} \rightarrow \mathbf{M} \quad (20.40)$$

があっても良い。実はこれまでも登場した逆元を作る演算子はまさに $\mathbf{M} \rightarrow \mathbf{M}$ という写像である。

またベクトルには，実数倍や回転といった写像がある。これらは実数のパラメタを一つとるので，ベクトル全体の集合を \mathbf{V} ，実数全体の集合を \mathbb{R} とし，実数のパラメタを r としたときに

$$\star_r :: (\mathbb{R} \times \mathbf{V}) \rightarrow \mathbf{V} \quad (20.41)$$

と書ける。例えば回転の場合は $\star_r = R_r$ である。

このようにとある集合（例えば $\mathbf{M} \times \mathbf{M}$ や $\mathbb{R} \times \mathbf{V}$ ）から異なる別な集合（例えば \mathbf{M} や \mathbf{V} ）へという写像を一般化するとどうなるだろうか。いま，集合 \mathbf{X} から集合 \mathbf{Y} への写像 f があり，集合 \mathbf{Y} から集合 \mathbf{Z} への写像 g が

あるとする．すなわち

$$f :: \mathbf{X} \rightarrow \mathbf{Y} \quad (20.42)$$

$$g :: \mathbf{Y} \rightarrow \mathbf{Z} \quad (20.43)$$

があるとする．また写像同士を二項演算子 \odot で合成できるものとする．例えば f と g の合成写像は \mathbf{X} を出発点に \mathbf{Y} を経由して \mathbf{Z} へと行くので

$$f \odot g :: \mathbf{X} \rightarrow \mathbf{Z} \quad (20.44)$$

と書ける．ここで合成演算子は結合性を満たすものとしておこう．

さらに

$$I_{\mathbf{X}} :: \mathbf{X} \rightarrow \mathbf{X}, I_{\mathbf{Y}} :: \mathbf{Y} \rightarrow \mathbf{Y}, I_{\mathbf{Z}} :: \mathbf{Z} \rightarrow \mathbf{Z} \quad (20.45)$$

という写像もあるとしよう．ここで

$$I_{\mathbf{Y}} \odot f = f \odot I_{\mathbf{X}} \quad (20.46)$$

とすると，写像 $I_{\mathbf{X}}$ と写像 $I_{\mathbf{Y}}$ はそれぞれ写像の合成演算子 \odot に対して単位元のように振る舞う．写像 g については

$$I_{\mathbf{Z}} \odot g = g \odot I_{\mathbf{Y}} \quad (20.47)$$

であるとする．このような写像 $I_{\mathbf{X}}, I_{\mathbf{Y}}, I_{\mathbf{Z}}$ を恒等写像と呼ぶ．

集合 \mathbf{X} ，集合 \mathbf{Y} ，集合 \mathbf{Z} の集合 $\mathbf{C} = \{\mathbf{X}, \mathbf{Y}, \mathbf{Z}\}$ と，写像 f と写像 g の集合 $\mathbf{P} = \{f, g\}$ と，写像合成演算子 \odot と，恒等写像の集合 \mathbf{I} の組み合わせ $(\mathbf{C}, \mathbf{P}, \odot, \mathbf{I})$ を圏と呼ぶ．写像合成演算子 (\odot) と恒等写像の集合 (\mathbf{I}) は自明であるためしばしば省略され，組み合わせ (\mathbf{C}, \mathbf{P}) を圏とする書き方もよくされる．

圏を考えると，変換や写像は全て射と呼ぶ決まりである．

いまある単位的半群 (\mathbf{M}, \star, O) があるとする．集合 \mathbf{C} を \mathbf{M} 及び $\mathbf{M} \times \mathbf{M}$ を元とする集合すなわち

$$\mathbf{C} = \{\mathbf{M}, \mathbf{M} \times \mathbf{M}\} \quad (20.48)$$

とし、集合 \mathbf{P} を \star のみを元とする集合すなわち

$$\mathbf{P} = \{\star\} \quad (20.49)$$

とすると、組み合わせ (\mathbf{C}, \mathbf{P}) は圏になっている。

—

圏の構造.

カーリー化された加算. カーリー化された \min .

対象, 射, 射の合成.

対象は整数集合.

射は整数から整数への関数.

恒等射.

射の合成. 関数の合成.

単位律. 結合律.

モノイドとは対象がひとつだけ存在する圏である. 四則演算は対象がひとつだけなので, モノイドである.

関手. カーリー化した圏から中置演算子への圏への写像.

20.4 余談：束*

20.5 この章のまとめ

	Totality	Associativity	Identity	Divisibility	Commutativity
Semicategory		✓			
Category		✓	✓		
Groupoid		✓	✓	✓	
Magma	✓				
Quasigroup	✓			✓	
Loop	✓		✓	✓	
Semigroup	✓	✓			
Monoid	✓	✓	✓		
Group	✓	✓	✓	✓	
Abelian Group	✓	✓	✓	✓	✓

表 20.3 代数的構造

	全域性	結合性	単位律	消約律	可換性
半圏 (semicategory)		✓			
圏 (category)		✓	✓		
亜群 (groupoid)		✓	✓	✓	
マグマ (magma)	✓				
擬群 (quasigroup)	✓			✓	
ループ (loop)	✓		✓	✓	
半群 (semigroup)	✓	✓			
モノイド (monoid)	✓	✓	✓		
群 (group)	✓	✓	✓	✓	
可換群 (Abelian group)	✓	✓	✓	✓	✓

第 21 章

モナドとプログラミング言語*

21.1 ジョイン*

21.2 クライスリ・トリプル*

21.3 *

21.4 余談：計算可能性*

μ 再帰関数

21.5 この章のまとめ*

第 22 章

ラムダ*

22.1 条件式

条件式は一種のシンタックスシュガーである．次のように関数 t, f, if を定義すると，それぞれ真，偽，条件分岐のように振る舞う．

$$t = \backslash xy \mapsto x \quad (22.1)$$

$$f = \backslash xy \mapsto y \quad (22.2)$$

$$\text{if } pxy = pxy \quad (22.3)$$

本当かどうか試してみよう．

$$\text{if } txy = txy \quad (22.4)$$

$$= (\backslash xy \mapsto x)xy \quad (22.5)$$

$$= x \quad (22.6)$$

$$\text{if } fxy = fxy \quad (22.7)$$

$$= (\backslash xy \mapsto y)xy \quad (22.8)$$

$$= y \quad (22.9)$$

確かに $\text{if } t$ は 1 番目の引数だけを， $\text{if } f$ は 2 番目の引数だけを残す．これは条件分岐そのものである．

22.2 整数*

22.3 Y コンビネータ*

22.4 余談：冗談言語*

22.5 この章のまとめ*

第 23 章

型付きラムダ*

23.1 型付きラムダ*

23.2 カリー＝ハワード同型対応*

23.3 *

23.4 *

C は公式にはラムダ式を採用していない．しかし C にラムダ式を持たせる拡張はいくつか提案されている．その一つが Apple 社が自社 OS の Grand Central Dispatch 機能のために行った「ブロック拡張」である．このブロック拡張を用いたクロージャの例を見てみよう．

C

```
#include <Block.h>
#include <stdio.h>
typedef int (^int_to_int)(int);
int_to_int make_plus_n(int n) {
    return Block_copy(^ (int x) {
        return n+x;
    });
}
int main(void) {
    int_to_int make_plus_10
        = make_plus_n(10);
    int x = make_plus_10(1);
    printf("x = %d\n", x);
    return 0;
}
```

コード中の `^(int x) { return n+x; }` がブロック（クロージャ）である。C のブロック拡張はオープンソースコミュニティに還元されているため、GCC や clang で使用可能である。

23.5 この章のまとめ*

第 24 章

マクロ*

24.1 Common Lisp のマクロ

24.2 Scheme のハイジェニックマクロ

24.3 C++ のテンプレート

C++14

```
auto lambda_exp  
    = [](auto x, auto y) { return \mXVar+\mYVar; };  
  
auto lambda_exp = [u = 1] { return u; };
```

複雑な構造に対する単純な操作について考える。

オブジェクト指向（クラス指向と呼ぶべきだが歴史はそうはならなかった）は、複雑な構造に対する単純な操作をうまく抽象化する。

例えば、複素数（実数に比べれば複雑だ）の足し算（単純だ）は、C89/90 ならば以下ようになる。

優秀な C プログラマならすぐに構造体と関数を導入するだろう。しかし、よりよい方法がある。C++ を使うことだ。（C99 ならば複素数を扱えるが、

言語の抽象度が C89/90 よりも高いわけではない。) C++98 ならば次のようにする。

複素数はあらかじめ用意されていたが、プログラマは独自の複雑な構造（行列だとか）を自前で作っておくことができる。複雑な構造に対する単純な操作を陽に扱えることは、特に GUI の設計、実装にとっては決定的となる。近代的な GUI は、複雑な構造（ビュー、コントローラ、モデルのそれぞれに当てはまる）の間をメッセージが単純に行き交うモデルであるからである。

（いま述べたのはオブジェクト指向のうちカプセル化についてだけである。オブジェクト指向の本当の力はポリモーフィズムにある。それについては後半で触れる。）

次は、単純な構造に対する複雑な操作について考える。

C++ のような素朴なオブジェクト指向機能だけでは、単純な構造に対する複雑な操作をうまく抽象化できない。次の Scheme のコードを見てもらいたい。

関数（Scheme では手続きと呼ぶ）`make-plus-n` は「引数に `n` を足す」という操作を作る。

とすると 5 が印刷される。数値（単純だ）に対する、飢えた足し算演算子（複雑な気分になる）を作ったのだ。C++98 で `make-plus-n` を作ることは可能であるが、簡潔とは言い難い。準備段階として標準関数オブジェクトクラステンプレート `std::plus` を使ってみる。

として作ったオブジェクト `plus2` は関数風に使える。例えば

は 5 を印字する。（あるいはより簡単に

と書いても同じことである。）もう一段の抽象化が C++ 版の `make-plus-n` である。

は 5 を印字する。驚くべきことに、優れた C++ プログラマは上のコードをさらさらと書く。

C++0x やアップルの Grand Central Dispatch (GCD) 対応版 C 言語では不格好ながらラムダ抽象が導入される。例えば C++0x では `make_plus_n`

は次のように書けるようになるはずである。

GCD では同様のラムダ式をこう書く。

さて、もう一度複雑な構造に対する単純な操作を振り返ってみよう。

オブジェクト指向の本質は関数のディスパッチである。メジャーなオブジェクト指向言語（C++ を含む）は単一ディスパッチ（第 1 引数が指す型ポインタによって実際の呼び出し先関数が決定される）であるが、LISP 用オブジェクトシステム CLOS（これは LISP 上に構築されている）は複数ディスパッチをサポートする。Clojure における例は「Closer to Clojure: ポリモーフィズム」から見てもらいたい。

もし、オブジェクト指向かクロージャ指向かのどちらかの言語を選べと言われたら、クロージャ指向を取ろう。どちらの言語でも、その言語の上にもう一方のパラダイムを築くことはできる。ただし、オブジェクト指向言語でクロージャ指向をサポートするのは骨の折れることだ。それに対し、クロージャ指向の言語でオブジェクト指向をサポートするのはたやすい。

—

24.4 余談：Template Haskell*

```
import Record
import Record.Lens
```

```
type Java = [r| { power :: Integer, url :: String } |]
type Link = [r| { title :: String, url :: String } |]
```

```
example :: Link
example = [r|{ title = "example", url = "http://www.example.org" }|]■
```

—

Lens

—

余談：C によるクロージャの実装*

簡単なクロージャの例として、引数に n を足す関数を生成する関数を C 言語で考える。やりたいことは Scheme で言えば

Scheme

```
(define (make-plus-n n) (lambda (x) (+ n x)))
```

なのだが、レキシカルクロージャを持たない C 言語では自前で変数をラップする必要がある。そこで、こんな構造体を作ってみる。

C

```
struct make_plus_n_context_t {
    int _n;
    int (*_func)
        (const struct make_plus_n_context_t *,
         int);
};
typedef struct make_plus_n_context_t
    MAKE_PLUS_N_CONTEXT_T;
```

次に、足し算関数の実体を用意しておく。

C

```
static int plus_n(
    const MAKE_PLUS_N_CONTEXT_T *context,
    int x) {
    return context->_n + x;
}
```

最後に、関数 `make_plus_n` を定義する。

```
MAKE_PLUS_N_CONTEXT_T *make_plus_n(int n) {  
    MAKE_PLUS_N_CONTEXT_T *context;  
    context = (MAKE_PLUS_N_CONTEXT_T *)  
        malloc(sizeof(MAKE_PLUS_N_CONTEXT_T));  
    context->n = n;  
    context->_func = plus_n;  
    return context;  
}
```

この関数は `make_plus_n.context_t` 構造体をメモリを新たに確保して返す。この構造体から `_func` を呼んでやるのは、次のようなマクロを用意すると便利である。

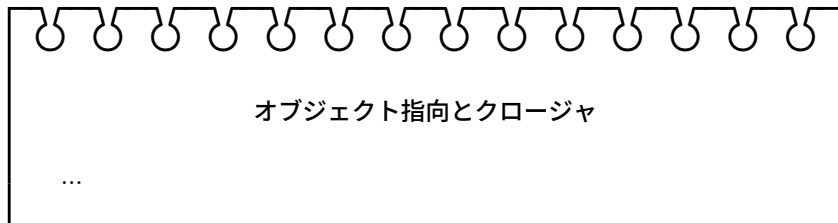
```
#define FUNC_CALL(context, param)  
    ((context)->_func((context), (param)))
```

本マクロは次のように使う。

```
int main(void) {  
    MAKE_PLUS_N_CONTEXT_T *plus_2  
        = make_plus_n(2);  
    int y = FUNC_CALL(plus_2, 1);  
    printf("%dn", y);  
    free(plus_2);  
    return 0;  
}
```

これが C 版クロージャの一例である。驚くべきことに C ウィザードはこのようなことは朝飯前にやってしまう。

24.5 この章のまとめ*



キーワード一覧

キーワード対訳表

参考文献

- [1] Haskell.org; <https://www.haskell.org/>, as of 2016.
- [2] 大角祐介: 新しい Linux の教科書; SB クリエイティブ, 2015.
- [3] 金谷 一朗: 「新しい Linux の教科書」を Mac で実践する;
<http://bit.ly/brew-on-mac>, 2016.
- [4] 金谷 一朗: ファンクション + アクション=プログラム; 工学社, 2011.
- [5] Miran Lipovaca: “Learn You a Haskell for Great Good: A Beginner’s Guide”; No Starch Press, 2011.
- [6] Benjamin C. Pierce: Types and Programming Languages; The MIT Press, 2002.
- [7] Ryan Lemmer: Haskell Design Patterns; Packt Publishing, 2015.
- [8] Bryan O’Sullivan: “Real World Haskell: Code You Can Believe In”; O’Reilly Media, 2008.
- [9] Graham Hutton: Programming in Haskell; Cambridge University Press; 2007.
- [10] Richard Bird: Thinking Functionally with Haskell; Cambridge University Press, 2014.
- [11] Paul Hudak, John Peterson, Joseph Fasel: A Gentle Guide to Haskell; <https://www.haskell.org/tutorial/index.html>, 1999.