

プログラマのための圏論

Scala Edition

Contains code snippets in Haskell and Scala

Bartosz Milewski 著

Igal Tabachnik 編

Isao Sonobe 訳

CATEGORY THEORY FOR PROGRAMMERS

Bartosz Milewski

Version dev

1980 年 1 月 1 日



This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License ([CC BY-SA 4.0](#)).

Converted from a series of blog posts by [Bartosz Milewski](#).

PDF and book compiled by [Igal Tabachnik](#).

Translated to Japanese by [Isao Sonobe](#).

L^AT_EX source code is available on GitHub:

<https://github.com/hmemcpy/milewski-ctfp-pdf>

<https://github.com/sonoisa/milewski-ctfp-pdf-japanese>

目次

編集者による注記	xiii
序文	xv
第 1 部	2
第 1 章 圈: 合成の本質	2
1.1 関数としての射	3
1.2 合成の性質	6
1.3 合成とはプログラミングの本質	10
1.4 チャレンジ	13
第 2 章 型と関数	14
2.1 誰が型を必要とするのか?	14
2.2 型とは合成可能性に関するこ	16

2.3	型とは？	18
2.4	なぜ数学的モデルが必要なのか？	22
2.5	純粋関数と汚い関数	26
2.6	型の例	27
2.7	チャレンジ	32
第 3 章	圏の大小	35
3.1	対象なし	35
3.2	単純なグラフ	36
3.3	順序	36
3.4	モノイドとしての集合	38
3.5	モノイドとしての圏	43
3.6	チャレンジ	47
第 4 章	Kleisli 圏	49
4.1	Writer 圏	54
4.2	Haskell での Writer	59
4.3	Kleisli 圏	63
4.4	チャレンジ	64
第 5 章	積と余積	66
5.1	始対象	67
5.2	終対象	69
5.3	双対性	71

5.4	同型	72
5.5	積	75
5.6	余積	84
5.7	非対称性	88
5.8	チャレンジ	92
5.9	参考文献	93
第 6 章	単純代数的データ型	94
6.1	積型	95
6.2	レコード	101
6.3	和型	104
6.4	型の代数	111
6.5	チャレンジ	117
第 7 章	関手	120
7.1	プログラミングにおける関手	123
7.1.1	Maybe 関手	124
7.1.2	等式推論	127
7.1.3	Optional	131
7.1.4	型クラス	133
7.1.5	C++ における関手	137
7.1.6	リスト関手	138
7.1.7	Reader 関手	141

7.2	関手としてのコンテナ	144
7.3	関手の合成	149
7.4	チャレンジ	153
第 8 章	関手性	154
8.1	双関手	154
8.2	積と余積の双関手	158
8.3	関手的な代数的データ型	161
8.4	C++ における関手	167
8.5	Writer 関手	171
8.6	共変関手と反変関手	174
8.7	プロ関手	179
8.8	ホム関手	182
8.9	チャレンジ	183
第 9 章	関数型	185
9.1	普遍構成	187
9.2	Curry 化	192
9.3	指数関数	197
9.4	デカルト閉圏	199
9.5	指数関数と代数的データ型	200
9.5.1	ゼロ乗	201
9.5.2	1 のべき乗	202

9.5.3	第1乗	202
9.5.4	和の指数関数	203
9.5.5	指数関数の指数関数	204
9.5.6	積の上の指数関数	204
9.6	Curry-Howard 同型	205
9.7	参考文献	209
第10章 自然変換		210
10.1	多相の関数	215
10.2	自然性を越えて	225
10.3	関手圏	229
10.4	2-圏	233
10.5	結論	238
10.6	チャレンジ	239
第2部		242
第11章 宣言的プログラミング		242
第12章 極限と余極限		251
12.1	極限と自然同型	257
12.2	極限の例	263
12.3	余極限	273
12.4	連続性	274

12.5	チャレンジ	278
第 13 章 自由モノイド		279
13.1	Haskell における自由モノイド	282
13.2	自由モノイドの普遍構成	284
13.3	チャレンジ	289
第 14 章 表現可能関手		291
14.1	ホム関手	293
14.2	表現可能関手	297
14.3	チャレンジ	306
14.4	参考文献	307
第 15 章 米田の補題		308
15.1	Haskell での米田	316
15.2	余米田	320
15.3	チャレンジ	321
15.4	参考文献	322
第 16 章 米田埋め込み		323
16.1	埋め込み	325
16.2	Haskell への応用	326
16.3	前順序の例	328
16.4	自然性	331

16.5	チャレンジ	332
第 3 部		334
第 17 章 すべては射に関することです		334
17.1	関手	334
17.2	可換図式	335
17.3	自然変換	336
17.4	自然同型	338
17.5	ホム集合	338
17.6	ホム集合同型	339
17.7	ホム集合の非対称性	341
17.8	チャレンジ	342
第 18 章 随伴		343
18.1	随伴と単位/余単位の対	344
18.2	随伴とホム集合	351
18.3	随伴からの積	358
18.4	随伴からの指数	363
18.5	チャレンジ	365
第 19 章 自由/忘却随伴		366
19.1	いくつかの直感	371
19.2	チャレンジ	375

第 20 章 モナド: プログラマによる定義	376
20.1 Kleisli 圈	379
20.2 魚の解剖学	384
20.3 do 記法	388
第 21 章 モナドと効果	397
21.1 問題点	398
21.2 解決策	399
21.2.1 部分性	400
21.2.2 非決定性	401
21.2.3 読み取り専用状態	406
21.2.4 書き込み専用状態	409
21.2.5 状態	411
21.2.6 例外	414
21.2.7 繙続	415
21.2.8 対話型入力	418
21.2.9 対話型出力	422
21.3 結論	424
第 22 章 モナドを圏論的に	425
22.1 モノイダル圏	432
22.2 モノイダル圏におけるモノイド	440
22.3 モナドとしてのモノイド	442

22.4	随伴からのモナド	444
第 23 章 余モナド		450
23.1	余モナドでのプログラミング	452
23.2	積型余モナド	453
23.3	合成の解剖	455
23.4	ストリーム余モナド	459
23.5	圏論的に余モナドを定義する	463
23.6	ストア余モナド	467
23.7	チャレンジ	472
第 24 章 F-代数		473
24.1	再帰	479
24.2	F-代数の圏	483
24.3	自然数	487
24.4	カタモルフィズム	488
24.5	畳み込み	491
24.6	余代数	494
24.7	チャレンジ	500
第 25 章 モナドに関する代数		501
25.1	T-代数	505
25.2	Kleisli 圈	509
25.3	余代数と余モナド	512

25.4	レンズ	513
25.5	チャレンジ	517
第 26 章 エンドと余エンド		518
26.1	双自然変換	521
26.2	エンド	523
26.3	エンドとイコライザとして	528
26.4	自然変換としてのエンド	530
26.5	余エンド	532
26.6	忍者米田の補題	537
26.7	プロ関手の合成	539
第 27 章 Kan 拡張		541
27.1	右 Kan 拡張	545
27.2	Kan 拡張と随伴	547
27.3	左 Kan 拡張	549
27.4	エンドとしての Kan 拡張	552
27.5	Haskell での Kan 拡張	555
27.6	自由関手	560
第 28 章 豊穣圏		564
28.1	なぜモノイダル圏か？	566
28.2	モノイダル圏	567
28.3	豊穣圏	569

28.4	前順序	572
28.5	距離空間	573
28.6	豊穣関手	575
28.7	自己豊穣化	577
28.8	2-圏との関連性	579
第 29 章 トポス		580
29.1	部分対象分類器	581
29.2	トポス	587
29.3	トポスと論理	588
29.4	チャレンジ	589
第 30 章 Lawvere 理論		590
30.1	普遍代数	590
30.2	Lawvere 理論	593
30.3	Lawvere 理論のモデル	597
30.4	モノイドの理論	599
30.5	Lawvere 理論とモナド	601
30.6	余エンドとしてのモナド	604
30.7	副作用の Lawvere 理論	609
30.8	チャレンジ	611
30.9	参考文献	612
第 31 章 モナド、モノイド、そして圏		613

31.1	双圈	614
31.2	モナド	620
31.3	チャレンジ	625
31.4	参考文献	625
付録		626
索引		626
謝辞		628
奥付		629
コピーレフトに関する通知		631

編集者による注記

これはプログラマのための圏論の Scala 版です。Bartosz Milewski 氏のブログ投稿シリーズを美しいタイプセットの PDF、さらにはハードカバーの書籍として利用できるようになり、非常に成功しています。本書を改善するために、誤字やエラーを修正したり、コードスニペットを他のプログラミング言語に翻訳したりする多数の貢献がありました。

私は、元の Haskell コードに続いて Scala コードを含む、この版の本を紹介することにわくわくしています。Scala のコードスニペットは、[Typelevel^{*1}](#)の貢献者によって寛大にも提供され、この本の形式に合わせて若干修正されています。

複数言語のコードスニペットをサポートするために、私は外部ファイルからコードスニペットを読み込むための LATEX マクロを使用しています。これにより、元のテキストをそのままにしながら、他の言語

^{*1} https://github.com/typelevel/CT_from_Programmers.scala

で本を簡単に拡張できます。このため、「Haskell で」というテキストを見たときは、心の中で「そして Scala でも」と言葉を付け足すべきです。

コードは以下のようにレイアウトされています: 元の Haskell コードに続いて Scala コードがあります。それらを区別するために、コードスニペットは、それぞれの言語のロゴ  および  の主要色を使用した縦棒で左側が囲まれています。例えば:

```
class Monoid m where
    mappend :: m -> m -> m
    mempty  :: m

trait Monoid[M] {
    def combine(x: M, y: M): M
    def empty: M
}
```

さらに、一部の Scala スニペットでは、部分的に適用された型に対してより良い構文をサポートする **Kind Projector**^{*2} コンパイラプラグインを使用しています。

^{*2} <https://github.com/non/kind-projector>

序文

私はしばらくの間、プログラマ向けの圏論の本を書くというアイディアを温めてきました。ここで言うプログラマとは、科学者ではなくエンジニア – 具体的にはコンピュータ科学者ではなく、現場の技術者を指します。これが狂気に聞こえるかもしれません、私は正気です。科学と工学の間には大きな隔たりがあるということは否定できませんが、私は常に物事を説明する強い衝動を感じてきました。Richard Feynman がシンプルな説明の達人であることに大きな敬意を払っています。私は Feynman ではありませんが、最善を尽くすつもりです。この序文を公開することで – これは読者に圏論を学ぶ動機を与えるためのものです – 議論を開始し、フィードバックを募ることを期待しています。^{*3}

^{*3} 私がこの素材をライブオーディエンスに教えるのを視聴することもできます。
<https://goo.gl/GT2UWU> で検索するか、「bartosz milewski category theory」で YouTube を検索してください。

私 は数段落のスペースを使って、この本があなたのために書かれたものであり、あなたが「豊富な余暇」に最も抽象的な数学分野の一つを学ぶべきではないというどんな反論も根拠がないことを納得させようと試みます。

私の楽観はいくつかの観察に基づいています。まず、圏論は非常に有用なプログラミングのアイデアの宝庫です。Haskell プログラマは長い間このリソースを利用してきましたが、そのアイデアはゆっくりと他の言語に浸透していますが、このプロセスは遅すぎます。私たちはそれを加速させる必要があります。

次に、数学には様々な種類があり、それぞれ異なる聴衆を惹きつけます。あなたは微積分や代数にアレルギーを持っているかもしれません、だからといって圏論を楽しめないわけではありません。私は圏論が特にプログラマの頭脳に適した数学であるとさえ主張したいです。それは圏論が – 特定のものではなく – 構造を扱うからです。プログラムを合成可能にする種類の構造を扱います。

合成は圏論の根本にあります – それは圏自体の定義の一部です。そして私は、プログラミングの本質が合成であると強く主張します。私たちは、優れたエンジニアがサブルーチンのアイデアを思いつくずっと前から、ずっと物事を合成してきました。しばらく前、構造化プログラミングの原理がプログラミングを革命的に変えました。それはコードブロックを合成可能にしました。次にオブジェクト指向プログラミングが登場しましたが、それはオブジェクトの合成に関するものです。関数型プログラミングは、関数と代数的データ構造の合成だけ

でなく、並行性の合成も可能にします。これは他のプログラミングパラダイムではほぼ不可能です。

第三に、私には秘密兵器、数学をプログラマにとってより消化しやすくするための肉切り包丁があります。あなたがプロの数学者であれば、仮定をきちんと整理し、言明を適切に述べて、証明を厳密に構築する必要があります。これにより、数学の論文や本は外部の人にとって非常に読みにくくなります。私は訓練を受けた物理学者であり、物理学では非公式な推論を使用して驚くべき進歩を遂げてきました。数学者は Dirac のデルタ関数を笑いましたが、それは偉大な物理学者 P. A. M. Dirac によって、いくつかの微分方程式を解くためにその場で作られました。彼らが Dirac の洞察を形式化した完全に新しい微積分の分野である超関数理論 (distribution theory) を発見したとき、彼らは笑うのをやめました。

もちろん、囁み碎いた説明を用いることで明らかに間違ったことを言うリスクがありますので、この本の非公式な議論の背後にはしっかりとした数学的理論があることを確かめるようにします。私は Saunders Mac Lane の *Categories for the Working Mathematician* (邦訳: *圏論の基礎*) の使い古されたコピーを持っています。

これは **プログラマのための圏論** であるため、私はすべての主要な概念をコンピュータコードを使用して説明します。あなたはおそらく、関数型言語がより一般的な命令型言語よりも数学に近いことを知っているでしょう。それはまた、より多くの抽象化力を提供します。したがって、圏論の恩恵を受ける前に Haskell を学ぶ必要があると言いた

くなるのは当然かもしれません。しかし、それでは圏論が関数型プログラミング以外に適用できないことを意味することになり、ただ単に正しくありません。したがって、私は多くの C++ の例を提供します。確かに、あなたはいくつかの醜い構文を克服する必要があります。パターンが冗長性の背景から際立たないかもしれませんし、高度な抽象化の代わりにコピー・アンド・ペーストを強いられるかもしれません、それは C++ プログラマの宿命です。

しかし、Haskell に関してあなたは無関係でいられません。Haskell プログラマになる必要はありませんが、C++ で実装されるアイデアをスケッチし、文書化する言語としてそれを必要とします。それはちょうど私が Haskell を始めた方法です。私は、その簡潔な構文と強力な型システムが C++ テンプレート、データ構造、およびアルゴリズムの理解と実装に大いに役立つことを発見しました。しかし、読者がすでに Haskell を知っていると期待することはできませんので、ゆっくりと導入し、進むにつれてすべてを説明します。

経験豊富なプログラマであれば、あなたは自分自身に尋ねているかもしれません：長い間、圏論や関数型の方法について心配することなくコーディングしてきましたが、何が変わったのでしょうか？ あなたは間違いなく、命令型言語に新しい関数型の機能が侵入してくる一貫した流れに気付いているはずです。オブジェクト指向プログラミングの砦である Java でさえ、ラムダを受け入れました。C++ は最近、急速なペースで進化しており – 数年ごとに新しい標準が出ています – 変化する世界に追いつこうとしています。このすべての動きは、破壊的

な変化、または私たち物理学者がそれを呼ぶように、相転移に備えるためのものです。あなたが水を加熱し続けると、最終的には沸騰を始めます。私たちは現在、ますます熱くなる水の中で泳ぎ続けるべきか、何らかの代替手段を探し始めるべきかを決定するカエルの立場にあります。

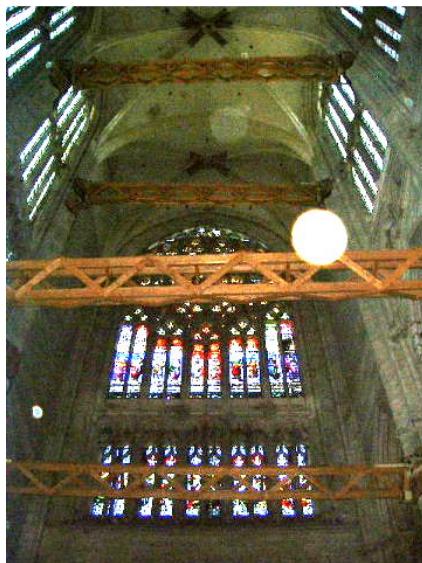


この大きな変化を推進する力の一つはマルチコア革命です。支配的なプログラミングパラダイムであるオブジェクト指向プログラミングは、並行性と並列性の領域で何の助けも与えてくれません。その代わりに、危険でバグのある設計を奨励します。オブジェクト指向の大前提であるデータの隠蔽は、データの共有と変更と組み合わせると、データ競合の原因となります。データを保護するためにロックを組み合わせるアイデアは良いですが、残念ながら、ロックは合成できませんし、ロックの隠蔽はデッドロックをより起こりやすくし、デバッグを困難にします。

しかし、並行性がなくても、ソフトウェアシステムの成長し続けている複雑さは、命令型パラダイムのスケーラビリティの限界を試しています。単純に言えば、副作用は手に負えなくなっています。副作用を持つ関数はしばしば便利で書きやすいものです。それらの効果は原則として、それらの名前やコメントにエンコードされます。`SetPassword` や `WriteFile` という関数は明らかに何らかの状態を変更し、副作用を生成していることがわかりますが、副作用を持つ関数を他の副作用を持つ関数の上に合成し、といったことを続けると、物事は複雑になります。副作用が根本的に悪いわけではありません – 隠された副作用が大規模な管理が不可能になる原因です。副作用はスケールしないし、命令型プログラミングとは副作用がすべてです。

ハードウェアの変化とソフトウェアの複雑さの増大は、プログラミングの基盤を再考することを強いています。ヨーロッパの偉大なゴシック大聖堂の建設者たちのように、私たちは物質と構造の限界に挑むために技術を磨いてきました。フランスのボーヴェにある未完成のゴシック大聖堂⁴は、この限界との深い人間的な闘いの証人です。それはすべての以前の記録を高さと軽さで打ち負かすことを意図していましたが、度重なる崩壊に見舞われました。鉄の棒や木製の支えが崩壊から守っていますが、明らかに多くの問題があります。現代の視点から見れば、近代的な材料科学、コンピュータモデリング、有限要素法、そして一般的な数学や物理学の助けなしに多くのゴシック建築

⁴ http://en.wikipedia.org/wiki/Beauvais_Cathedral



ボーヴェ大聖堂の崩壊を防ぐための場当たり的な措置

が成功裏に完成したことは奇跡です。私は将来の世代が、複雑なオペレーティングシステム、ウェブサーバー、そしてインターネットインフラストラクチャを構築するために私たちが見せているプログラミング技術に同じように感心することを望みます。そして、正直に言って、彼らはそうすべきです。なぜなら、私たちは非常に脆弱な理論的基盤に基づいてこれらを行ってきたからです。私たちは前進したいのであれば、それらの基盤を修正する必要があります。

第 1 部

1

圈: 合成の本質

圈 は驚くほど単純な概念です。圈 (category) は対象 (object) とそれらの間を結ぶ矢印 (arrow) で構成されます。そのため圈はとても図示しやすいです。対象は円や点として描かれ、矢印は...矢印です。(たまに変わり種として、対象を豚として、矢印を花火として描くこともあります。) しかし、圈の本質は合成 (composition) です。合成の本質が圈であると言ってもいいです。矢印は合成できるので、対象 A から B への矢印があり、さらに B から C への矢印があるならば – それらの合成である – A から C への矢印が存在しなければなりません。



図では、 A から B への矢印と、 B から C への矢印があるならば、それらの合成である A から C への直接の矢印も存在しなければなりません。この図には、恒等射(後述)が欠けているため完全な図ではありません。

1.1 関数としての射

抽象的すぎてナンセンスに感じますか？落胆しないでください。具体例で考えましょう。矢印、つまり射(*morphism*)を関数と考えてみましょう。型 A の引数を取り、 B を返す関数 f があります。そして、 B を取り、 C を返す別の関数 g があります。それらを、 f の結果を g に渡すことで合成することができます。これにより、 A を取り、 C を返す新しい関数が得られます。

数学では、このような合成は関数の間に小さな円を置いて表されます: $g \circ f$ 。合成の順番が右から左であることに注意してください。この順序が混乱の原因となることもあります。Unix のパイプ記法

```
lsof | grep Chrome
```

もしくは、F#の chevron 記法`>>`はご存知でしょうか。これらはどちらも左から右に処理が進みます。しかし、数学や Haskell では関数は右から左に合成されます。 $g \circ f$ を「 f の後に g 」と読むと理解しやすいかもしれません。

さらに具体的に C 言語のコードを書いてみましょう。型 A の引数を取り、型 B の値を返す関数 `f` があるとします：

```
B f(A a);
```

他にももう一つ：

```
C g(B b);
```

これらの合成は次のようになります：

```
C g_after_f(A a)
{
    return g(f(a));
}
```

ここでも再び、`g(f(a))` という右から左への合成が見られます。今回は C 言語です。

残念ながら、C++ 標準ライブラリには二つの関数を取ってその合成を返すテンプレートがありません。そのため、少し Haskell で試してみましょう。A から B への関数の宣言は以下の通りです：

```
f :: A -> B
```

```
val f: A => B
```

同様に:

```
g :: B -> C
```

```
val g: B => C
```

これらの合成は:

```
g . f
```

```
g compose f
```

Haskell のシンプルさを知ると、C++ で端的に関数概念を表現できな
いことに少し恥ずかしくなるでしょう。実際、Haskell では Unicode
文字を使用して合成を書くことができます:

```
g ° f
```

さらに、Unicode の二重コロンと矢印も使用できます:

```
f :: A → B
```

ここで最初の Haskell のレッスンです。二重コロンは「...の型を持つ」という意味です。関数型は二つの型の間に矢印を挿入することで作成されます。二つの関数を合成するには、それらの間にピリオド(または Unicode の円)を挿入します。

1.2 合成の性質

任意の圏において合成が満たさなければならない二つの非常に重要な性質があります。

1. 合成が結合的であること。三つの射、 f 、 g 、および h が合成できる(つまり、それらの対象が初めから終わりまで一致する)場合、それらを合成するために括弧は不要ということ。数学的な記法では、これは以下のように表されます:

$$h \circ (g \circ f) = (h \circ g) \circ f = h \circ g \circ f$$

(擬似) Haskell では以下のようにになります:

```
f :: A -> B
g :: B -> C
h :: C -> D
h . (g . f) == (h . g) . f == h . g . f
```

```

val f: A => B
val g: B => C
val h: C => D

h compose (g compose f) === (h compose g) compose f ===
↪ h compose g compose f

```

(「擬似」と言ったのは、関数に対する等価性は定義されていないからです。)

関数を扱う場合、結合則はかなり明白ですが、他の圏ではそうとは限りません。

- 任意の対象 A に対して、合成の単位となる射が存在すること。この射は対象自体をループします。合成の単位であるとは、 A から始まるか、または A で終わるいかなる射と合成しても、同じ射を返すということです。対象 A の単位射は id_A (A 上の恒等射 (*identity*)) と呼ばれます。数学的な記法では、 f が A から B へ行く場合:

$$f \circ \text{id}_A = f$$

そして

$$\text{id}_B \circ f = f$$

関数を扱う場合、恒等射はその引数をそのまま返す恒等関数として実装されます。この実装はすべての型に対して同じであり、これはこの

関数が全称多相的であることを意味します。C++ ではテンプレートとして定義することができます:

```
template<class T> T id(T x) { return x; }
```

もちろん、C++ では単純にはいかず、何を渡すかだけでなく、どのように渡すかも考慮に入れなければなりません(つまり、値として、参照として、const 参照として、ムーブとしてなど)。

Haskell では、恒等関数は標準ライブラリ (Prelude と呼ばれる) の一部です。これがその宣言と定義です:

```
id :: a -> a  
id x = x
```

```
def identity[A](a: A): A = a
```

ご覧の通り、Haskell では多相的関数は簡単です。宣言において、型を型変数に置き換えるだけです。ポイントは、具体的な型の名前は常に大文字で始まり、型変数の名前は小文字で始まることです。ここでは `a` はすべての型を表しています。

Haskell の関数定義は関数名に続けて形式パラメータが続きます — ここではただ一つ、`x` です。関数の本体は等号に続きます。この簡潔さは初心者にとってしばしば衝撃ですが、すぐにそれが完全に理にかなっていることがわかります。関数の定義と関数の呼び出しは関数型プログラミングの中心であるため、その構文は最小になるよう簡略化

されています。引数リストの周りに括弧がないだけでなく、引数間にコンマがないことにも気づくでしょう（後で、複数の引数を持つ関数を定義するときにそれを見ることになります）。

関数の本体は常に式です — 関数内には文がありません。関数の結果はこの式です — ここでは、単に `x` です。

これで第二の Haskell レッスンは終了です。

恒等性に関する条件は（再び、擬似 Haskell で）以下のように書かれます：

```
f . id == f
id . f == f

f compose identity[A] == f
identity[B] _ compose f == f
```

あなたは疑問を持ったかもしれません。何もしない恒等関数を一体誰が気にするのだろうかと。では、なぜ私たちはゼロという数字を気にするのでしょうか？ ゼロは無の象徴です。古代ローマ人はゼロのない数字システムを持っていましたが、それでも彼らは優れた道路や水道を建設しました。そのいくつかは今日まで残っています。

記号的な変数を扱うとき、ゼロや `id` のような中立的な値は非常に便利です。それが、代数得意としなかったローマ人と、ゼロの概念に精通していたアラブ人やペルシャ人の違いです。そのように、高階関数への引数として、または返り値として、恒等関数は非常に便利で

す。高階関数は関数の記号的操作を可能にします。関数の代数ということです。

まとめると、圏は対象と矢印(射)で構成されます。射は合成でき、その合成は結合的です。すべての対象には合成の単位となる恒等射があります。

1.3 合成とはプログラミングの本質

関数型プログラマは問題に対して独特の方法でアプローチします。彼らは非常に禅的な質問から始めます。例えば、対話型プログラムを設計する際には、対話とは何なのかと問います。Conway のライフゲームを実装する際には、生命の意味について考えるでしょう。この精神で、私たちはプログラミングとは何かという問いを立てます。最も基本的なレベルでは、プログラミングとはコンピュータにすべきことを教えることです。「メモリアドレス x の内容を EAX レジスタの内容に加えてください。」のように。しかし、たとえアセンブリ言語でプログラミングしていても、私たちがコンピュータに指示する命令は、より意味のある何かの表現です。私たちは自明ではない問題を解決しています(もし、それが些細な問題であれば、コンピュータの助けは必要ないでしょう)。では、そういう問題はどのようにして解決するでしょうか？ 私たちは、大きな問題をより小さな問題に分解します。もし小さな問題がまだ大きすぎる場合は、さらに分解を続けます。そして最終的に、すべての小さな問題を解決するコードを書きます。そし

て、プログラミングの本質が現れます：それらのコードの断片を組み合わせて、より大きな問題の解決策を作り上げます。分解は、断片から元を復元できなければ意味がありません。

この階層的な分解と再構成（合成）のプロセスは、コンピュータによって私たちに課されたものではありません。それは人間の頭脳の限界を反映しています。私たちの脳は、一度に扱える概念の数が限られています。心理学で最も引用される論文の一つである「[The Magical Number Seven, Plus or Minus Two](#)¹」は、私たちが心に留めることができる情報の「チャンク」の数は 7 ± 2 であると主張しています。人間の短期記憶に対する理解の詳細は変わるかもしれません、それが限られていることは確かです。要するに、私たちにはオブジェクトのスープやコードのスパゲッティを処理することはできません。私たちは構造を必要とするのです。それは、うまく構造化されたプログラムが見た目に楽しいからではなく、そうでなければ私たちの脳が効率的に処理できないからです。私たちはしばしばあるコードの断片をエレガントあるいは美しいと表現しますが、本当の意味は、それが私たちの限られた人間の頭脳にとって消化しやすいということです。エレガントなコードは、私たちの精神的な消化システムに取り込むのにちょうど良いサイズと数のチャンクを作り出します。

¹ http://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

では、プログラムの構成に適したチャンクとは何でしょうか？その表面積は体積より遅く増加する必要があります。(私はこのアナロジーが好きです。幾何学的物体の表面積はそのサイズの 2 乗で増加し、これは体積の増加がサイズの 3 乗で増加することに比べてより進みが遅いという直感があります。)ここで表面積とは、チャンクを合成するために必要な情報です。体積とは、それらを実装するために必要な情報です。このアイディアは、一旦チャンクが実装されたら、その実装の詳細を忘れることができ、他のチャンクとどのように相互作用するかに集中できるということです。オブジェクト指向プログラミングでは、表面はオブジェクトのクラス宣言やその抽象インターフェースです。関数型プログラミングでは、それは関数の宣言です。(少し単純化していますが、それが要点です。)

圏論は極端な意味で、私たちに対象の内部を見るなどを積極的に抑制します。圏論の対象は抽象的で曖昧な実体です。あなたがそれについて知り得るすべては、それがどのように他の対象と関係しているか、つまりそれが射を使ってどのように接続しているかだけです。これはインターネット検索エンジンが、インバウンドリンクとアウトバウンドリンクを分析することによってウェブサイトをランク付けする方法と似ています(彼らが不正を働かない限り)。オブジェクト指向プログラミングでは、理想化されたオブジェクトはその抽象インターフェース(純粋な表面、体積なし)を通じてのみ可視化され、メソッドは射の役割を果たします。オブジェクトの実装を掘り下げて、他のオブジェ

クトとどのように組み合わせるかを理解する必要がある瞬間に、あなたはプログラミングパラダイムの利点を失います。

1.4 チャレンジ

1. あなたのお気に入りの言語で恒等関数を、できる限り良く実装してください（もし Haskell があなたのお気に入りの言語であれば、2 番目に好きな言語で試してください）。
2. あなたのお気に入りの言語で合成関数を実装してください。それは 2 つの関数を引数として取り、それらの合成を返す関数です。
3. あなたの合成関数が恒等性条件を遵守することをテストするプログラムを書いてください。
4. ワールド・ワイド・ウェブはいかなる意味で圏ですか？ リンクは射ですか？
5. 人々を対象とし、友人関係を射とした場合、Facebook は圏ですか？
6. 有向グラフはどういう場合に圏になりますか？

2

型と関数

型と関数の圈はプログラミングにおいて重要な役割を果たします。そこで、型とは何か、そしてなぜ必要なのかについて話しましょう。

2.1 誰が型を必要とするのか？

型付けにおける、静的 vs. 動的、強い vs. 弱いに関しては、その利点を巡って様々な議論があります。これらの選択を一つの思考実験で説明しましょう。何百万匹の猿がコンピュータのキーボードでランダムにキーを打ち、プログラムを生成し、コンパイルし、そして実行する姿を想像してください。



マシン語では、猿によって生成されたバイトの任意の組み合わせが受け入れられ、実行されます。しかし、高級言語を使うと、ありがたいことにコンパイラが字句的、文法的なエラーを検出してくれます。多くの猿はバナナを失いますが、残ったプログラムは有用である可能性が高くなります。型チェックは、意味を成さないプログラムに対するもう一つの障壁を提供します。さらに、動的型付け言語では型の不一致が実行時に発見される一方、強い静的型チェックが行われる言語では、型の不一致がコンパイル時に発見され、実行される前に多くの不正なプログラムが排除されます。

したがって、問題は私たちが猿を喜ばせたいのか、それとも正しいプログラムを作りたいのかということです。

猿のタイピング実験での通常の目標はシェイクスピアの全作品を生み出すことです。スペルチェッカーや文法チェッカーが実験ループの中にあれば、勝算が劇的に上がります。型チェッカーの類似物は、ロミオが人間であると宣言されたら、彼が葉を生やしたり、彼の強力な

重力場が光子を閉じ込めたりしないこと確認することで、さらにその上を行くでしょう。

2.2 型とは合成可能性に関するこ

圈論は射の合成に関する理論です。しかし、どの二つの射も合成できるわけではありません。一つの射のターゲット対象が次の射のソース対象と同じでなければなりません。プログラミングでは、一つの関数の結果を別の関数に渡します。ターゲット関数がソース関数によって生成されたデータを正しく解釈できない場合、プログラムは動作しません。両端がマッチしなければ合成は機能しません。言語の型システムが強ければ強いほど、このマッチを記述し、機械的に検証できるようになります。

強い静的型チェックに対する唯一のまともな反論は、それが意味的に正しいプログラムを排除してしまう可能性があるということです。実際には、これは極めてまれにしか起こりませんし、いずれにせよ、どんな言語も型システムを迂回する何らかの方法を提供しています。Haskell さえも `unsafeCoerce` があります。しかし、このような手段は慎重に使用するべきです。Franz Kafka のキャラクター、Gregor Samsa は、巨大な虫に変身するときに型システムを破壊し、その結果は皆さんも知つての通りです。

もう一つよく耳にする反論は、型付けがプログラマに多大な負担を課すというものです。私自身、C++ でイテレータの宣言をいくつか書

かなければならなかった後、この感情に同情することができました。ただし、**型推論**と呼ばれる技術があり、コンパイラは、ほとんどの型をそれらが使われている文脈から推論することができます。C++ では、変数を `auto` と宣言し、コンパイラにその型を決定させることができます。

Haskell では、まれな場合を除き、型アノテーションの記載は完全に任意です。それでも、プログラマは書く傾向があります。なぜなら、それらはコードの意味について多くを語り、コンパイルエラーを理解しやすくするからです。Haskell ではプロジェクトを始めるときにまず型の設計を行うのが一般的慣習になっています。その後、型アノテーションが実装を駆動し、コンパイラによって強制されるコメントになります。

強い静的型付けは、コードをテストしない言い訳としてよく使われます。時々、Haskell のプログラマによる「コンパイルが通ったのなら、それは正しいに違いない」という発言を耳にしているかもしれません。もちろん、型が正しいということをもって、正しい出力を生み出す、正しいプログラムであることが保証されるわけではありません。この無謀な態度の結果、複数の研究において、Haskell が一般に期待されるほどコードの品質で他の言語よりも大きく抜きん出ているわけではないことが判りました。営利目的の場面では、バグ修正のプレッシャーは一定の品質レベルまでに留まるようです。その品質レベルはすべてソフトウェア開発の経済性とエンドユーザーの許容範囲に関係するものであり、プログラミング言語や方法論とはほとんど関係があ

りません。より良い基準は、予定よりも遅れたプロジェクトの数、または機能が大幅に減った状態で納品されたプロジェクト数を測定することでしょう。

単体テストが強い型付けを置き換えることができるという議論については、強い型付け言語での一般的なリファクタリングのプラクティスを考えてみましょう。題材は、特定の関数の引数の型を変更することです。強い型付け言語では、その関数の宣言を変更し、その後、ビルドが通らなかったすべての部分を修正するだけで済みます。弱い型付け言語では、関数が前とは異なるデータを期待するようになった事実を呼び出し元に伝播させることができません。単体テストは一部の不一致を捉えるかもしれません、テストはほぼ常に確率的であり、決定的ではありません。証明の代用にするにはテストは貧弱です。

2.3 型とは？

型の最もシンプルな直感的理解は、それらが値の集合であるということです。型 `Bool` (Haskell では具体的な型は大文字で始まることを思い出してください) は、二つの要素 `True` と `False` からなる集合です。型 `Char` は、`a` や `ä` のようなすべての Unicode 文字の集合です。

集合は有限であることも無限であることもあります。型 `String`、これは `Char` のリストの別名ですが、無限集合の例です。

`x` を `Integer` と宣言するとき:

```
x :: Integer
```

```
val x: BigInt
```

これは整数集合の要素であると言っているのです。`Integer` は Haskell では無限集合であり、任意精度の算術を行うために使用できます。マシン型に対応する有限集合 `Int` もあります。これは C++ の `int` のようなものです。

型と集合のこの同一視にはいくつかの微妙な問題があります。循環定義を含む多相的関数や、すべての集合の集合は手に入らないという事実に関する問題です。しかし、約束しましたが、私は数学について厳密であることはありません。素晴らしいことに、`Set` と呼ばれる集合の圏があり、私たちはそれを使います。`Set` では、対象は集合であり、射(矢印)は関数です。

`Set` は非常に特別な圏です。なぜなら、実際にその対象の中を覗いて多くの直感を得ることができるからです。例えば、空集合には要素がないことを知っています。特別な単集合（一つの要素のみからなる集合）があることも知っています。関数が一つの集合の要素を別の集合の要素にマップすることを知っています。それは二つの要素を一つにマップすることはできますが、一つの要素を二つにマップすることはできません。恒等関数が集合の各要素をそれ自身にマップすることを知っています、などなど。計画は徐々にこれらすべての情報を忘れ

ていき、その代わりにすべての概念を純粋に圏論的な用語、つまり対象と射の観点から表現することです。

理想的な世界では、Haskell の型は集合であり、Haskell の関数は集合間の数学的関数であると単純に言えばいいのですが、ひとつだけ小さな問題があります。数学的な関数はコードを実行しません — 答えを知っているだけです。Haskell 関数は答えを計算しなければなりません。答えが有限ステップ数で得られる場合は問題ありません — その数がいかに大きくとも。しかし、再帰を含む計算があり、それらは決して停止しない可能性があります。Haskell から停止しない関数を取り除くことはできません。なぜなら、停止する関数と停止しない関数を区別することは決定不能だからです — 有名な停止問題です。そのため、コンピュータ科学者は、ボトムと呼ばれ `_|_` や Unicode 文字 `⊥` で書かれる特別な値で型を拡張するという天才的な、または視点によつては大胆なアイデアを思いつきました。この「値」は、停止しない計算に対応します。従って、関数が次のように宣言されている場合:

```
f :: Bool -> Bool  
val f: Boolean => Boolean
```

これは `True` か `False`、または `_|_` を返すかもしれません。`_|_` は、これが決して停止しないことを意味します。

興味深いことに、ボトムを型システムの一部として受け入れるなら、すべてのランタイムエラーをボトムとして扱い、また、関数が明示的

にボトムを返すことを許可するようにすると便利です。後者は通常、次のように式 `undefined` を使用して行われます:

```
f :: Bool -> Bool  
f x = undefined  
  
val f: Boolean => Boolean = x => ???
```

この定義は型チェックを通過します。なぜなら、`undefined` はボトムに評価され、そして、ボトムは `Bool` を含む任意の型のメンバーであるからです。さらに次のように書くこともできます:

```
f :: Bool -> Bool  
f = undefined  
  
def f: Boolean => Boolean = ???
```

(`x` がありません) なぜなら、ボトムは型 `Bool -> Bool` のメンバーでもあるからです。

ボトムを返す可能性のある関数は部分関数と呼ばれ、すべての可能な引数に対して正式な結果を返す全関数と対比されます。

ボトムがあるため、Haskell の型と関数の圏は `Set` ではなく `Hask` として言及されることがあります。理論的な観点から、これは終わることのない複雑さの源です。ですので、この点において私は肉切り包丁を使い、この議論の筋を切り落とします。実用的な観点からは、停

止しない関数とボトムを無視し、**Hask** を正当な **Set** として扱っても構いません。^{*1}

2.4 なぜ数学的モデルが必要なのか？

プログラマとして、あなたは使用しているプログラミング言語の構文と文法に精通しているでしょう。言語のこれらの側面は通常、言語仕様書の一番最初に形式的記法を用いて記述されています。しかし、言語の意味、すなわち意味論を記述することははるかに困難です。それには多くのページを必要とし、十分に形式的であることは稀で、大体は不完全です。そのため、プログラミング言語に精通した経験豊富なエンジニアの間で終わりのない議論が交わされ、言語標準の細かな点の解釈に特化した本という小規模な産業が存在しています。

言語の意味論を記述するための形式的ツールがありますが、その複雑さのため、実際に使われている巨大なプログラミング言語ではなく、主に簡略化された学術研究用言語に対して使用されています。そのようなツールの一つである**操作的意味論** (*operational semantics*) は、プログラム実行のメカニズムについて記述します。それは、形式化・理想化されたインタープリタを定義します。C++ のような産業的な言

^{*1} Nils Anders Danielsson, John Hughes, Patrik Jansson, Jeremy Gibbons, **Fast and Loose Reasoning is Morally Correct**. この論文は、ほとんどの文脈でボトムを無視することを正当化してくれます。

語の意味論は、通常、「抽象機械」という名前の非形式的な操作的推論を使用して記述されます。

問題は、操作的意味論を使用してプログラムに関する証明を行うのが困難であるということです。プログラムが持つ性質を示すためには、理想化されたインタープリタを用いて本質的に「実行」しなければなりません。

プログラマが正確性の形式的証明を決してしないことに問題はありません。私たちは常に自身が正しいプログラムを書いていると「思って」います。キーボードの前に座って、「さて、数行のコードを投げてみて何が起こるか見てみよう」と言う人はいません。私たちは書いたコードが望ましい結果を生み出す特定のアクションを実行すると思っています。それがそうでない場合、私たちは普通とても驚きます。つまり、私たちは書いたプログラムについて常に推論しているのです。そして、私たちは通常それを頭の中でインターパリタを実行することによって行なっています。ただ、すべての変数を追跡するのは本当に難しいです。コンピュータはプログラムの実行が得意ですが、人間はそうではありません！もし私たちがそうだったら、私たちはコンピュータを必要としませんでした。

しかし、代わりがあります。それは表示的意味論 (*denotational semantics*) と呼ばれ、数学に基づいています。表示的意味論では、プログラミングのすべての構成に対してその数学的解釈が与えられます。それにより、プログラムの特性について証明したければ、数学の定理を証明するだけです。定理の証明は難しいと思うかもしれません、

しかし実際は、何千年もの間、人類は数学的方法を作り上げてきたので、利用できる知識がたくさん蓄積されています。また、プロの数学者が証明するような定理に比べれば、プログラミングで遭遇する問題は、自明ではないにせよ、通常極めて単純なものであることがほとんどです。

Haskellにおける階乗関数の定義を考えてみましょう。Haskellは表示的意味論にとても適する言語です：

```
fact n = product [1..n]
```

```
val fact = (n: Int) => (1 to n).product
```

式 `[1..n]` は 1 から `n` までの整数のリストです。関数 `product` はリストのすべての要素を掛け合わせます。これは数学の教科書から持ってきた階乗の定義そっくりです。これと C 言語のものを比較してみましょう：

```
int fact(int n) {
    int i;
    int result = 1;
    for (i = 2; i <= n; ++i)
        result *= i;
    return result;
}
```

これ以上、何か言う必要はありますか？

オーケー、これは安直な一撃だったことは先に認めましょう！ 階乗関数には明らかな数学的表現があります。鋭い読者は尋ねるかもしれません。キーボードから文字を読み取る、またはネットワークを介してパケットを送信するといったものの数学的モデルは何ですかと。長い間、それはおそらくかなり複雑な説明につながる厄介な質問でした。表示的意味論は、便利なプログラムを書くために不可欠な多くの重要なタスクには最適ではないようであり、そして、それらは操作的意味論では簡単に扱うことができました。その突破口は圏論から来ました。Eugenio Moggi は、計算効果をモナドにマッピングできることを発見しました。これは、表示的意味論に新たな命を吹き込み、純粹関数型プログラムをより使いやすくするだけでなく、伝統的なプログラミングにも新たな光を当てる重要な発見でした。もっと圏論の道具を導入した後に、モナドについて話すことにします。

プログラミングに関する数学的モデルを持つことの重要な利点の一つは、ソフトウェアの正確性の形式的証明を行えるということです。これは、消費者向けソフトウェアを書いているときはそれほど重要ではないかもしれません、プログラミングの領域には失敗の代償が法外に大きい場合や、人命に関わる場合があります。そこまででなくとも、あなたは医療システム用 Web アプリケーションを書くときに、Haskell 標準ライブラリの関数やアルゴリズムは正確性の証明付きであるという考えに感謝するかもしれません。

2.5 純粋関数と汚い関数

C++ や他の命令型言語で私たちが関数と呼ぶものは、数学者が関数と呼ぶものと同じではありません。数学的な関数は単に値から値への写像です。

プログラミング言語で数学的な関数を実装できます。そのような関数は、入力値が与えられると出力値を計算します。数の二乗を生成する関数はおそらく入力値をそれ自身で乗算します。それは呼ばれるたびに、そして同じ入力で呼ばれるたびに、同じ出力を生成することが保証されます。月の位相と共に数の二乗が変わることはありません。

また、数の二乗を計算することが、あなたの犬に美味しいおやつを配るといった副作用を持っているべきではありません。そのような「関数」は数学的な関数として簡単にモデル化することができません。

プログラミング言語で、同じ入力が与えられたときに常に同じ結果を生成し、副作用がない関数は**純粋関数** (*pure function*) と呼ばれます。Haskell のような純粋関型言語では、すべての関数が純粋です。そのため、これらの言語に表示的意味論を与えて、圏論を使ってモデル化することは容易です。他の言語については、常に、自身を純粋なサブセットに制限したり、副作用を分けて推論したりすることができます。後ほど、モナドがどのようにして純粋関数のみを使用してあらゆる種類の副作用をモデル化できるかを見ていきます。従って、実際には、数学的な関数に自身を制限することで失うものは一切ありません。

2.6 型の例

一度、型が集合であると思うと、かなり風変わりな型を考えることができます。例えば、空集合に対応する型は何でしょうか？ いいえ、それは C++ の `void` ではありません。ですが、この型は Haskell では `Void` と呼ばれています。これはどんな値も持たない型です。あなたは `Void` を受け取る関数を定義することはできますが、それを決して呼び出すことはできません。それを呼び出すためには、`Void` 型の値を提供しなければならず、そんなものは存在しません。この関数が返すことができるものに制限はまったくありません。任意の型を返すことができます（ただし、それは決して呼び出されないため、実際には返すことはありません）。言い換えれば、それは返り値の型において多相的な関数です。Haskell では次のように名付けています：

```
absurd :: Void -> a  
  
def absurd[A]: Nothing => A
```

（思い出してください。`a` は任意の型を表す型変数です。）この名前（和訳：不合理）は偶然ではありません。型と関数のより深い解釈が、Curry-Howard 対応と呼ばれる論理の観点からあります。型 `Void` は偽を表し、関数 `absurd` の型は、ラテンの格言「*ex falso sequitur quodlibet*」、つまり「偽からは何でも導ける」に対応します。

次は、単集合に対応する型です。それはただ 1 つ可能な値を持つ型です。値の意味はただ「存在する」のみです。すぐにはそう思えないかもしれません、これは C++ の `void` です。この型からとこの型への関数を考えてみてください。`void` からの関数は常に呼び出すことができます。純粋関数であれば、常に同じ結果を返します。下記はそのような関数の例です：

```
int f44() { return 44; }
```

あなたはこの関数が「無」を受け取ると思うかもしれません、私たちがさっき見たように、「無」を受け取る関数は、「無」を表す値が存在しないため、決して呼び出すことができません。では、この関数は何を受け取るのでしょうか？ 概念的には、何か唯一のインスタンスであればよいダミー値を受け取ります。そのため、私たちはその値について明示的に言及する必要はありません。しかし、Haskell ではこの値に対する記号があります。空の丸括弧 () です。面白い偶然で（それとも偶然ではないのでしょうか？）、`void` 型の関数の呼び出しは C++ と Haskell で同じように見えます。また、Haskell の簡潔さを愛するために、同じ記号 () が、単集合に対応する型、コンストラクタ、および唯一の値を表すために使用されます。そして、Haskell では関数はこうなります：

```
f44 :: () -> Integer  
f44 () = 44
```

```
val f44: Unit => BigInt = _ => 44
```

最初の行は、**f44** が「Unit」と発音する型 () から **Integer** 型への関数であるという宣言です。2 行目は **f44** の定義であり、Unit 型の唯一のコンストラクタ、つまり () にパターンマッチをして、数値 44 を生成します。この関数を呼び出すには Unit 値 () を与えます:

```
f44 ()
```

Unit を受け取るすべての関数は、ターゲット型から单一の要素を選ぶことと同等であることに注意してください(ここでは、**Integer** 44 を選ぶこと)。実際、**f44** を数 44 の異なる表現と考えることもできます。これは、集合の要素について直接言及する代わりに、関数(射)に関する話に置き換える方法の一例です。Unit 型から任意の型 A への関数は、集合 A の要素と一对一の対応関係にあります。

返り値が **void** 型の関数、もしくは Haskell で返り値が Unit 型の関数はどうでしょうか？C++ ではそのような関数は副作用のために使用されますが、これらが数学的な意味での本当の関数ではないことはわかっています。Unit を返す純粋な関数は何もせず、引数を破棄します。

数学的には、集合 A から单集合への関数は、A のすべての要素をその单集合の单一要素に写像します。すべての A に対して、そのような関数はただ一つだけ存在します。これは **Integer** 用のその関数です：

```
fInt :: Integer -> ()  
fInt x = ()  
  
val fInt: BigInt => Unit = x => ()
```

これに任意の整数を与えると、Unit が返ってきます。簡潔さの精神から、Haskell ではワイルドカードパターンであるアンダースコアを、捨てられる引数に対して使用することができます。この方法により名前を考えずに済みます。そのため、上記は次のように書き換えることができます:

```
fInt :: Integer -> ()  
fInt _ = ()  
  
val fInt: BigInt => Unit = _ => ()
```

この関数の実装は、それに渡された値に依存しないだけでなく、引数の型にも依存しないことに注意してください、

任意の型に対して同じ方程式で実装できる関数は、パラメトリック多相性を持つと言われます。あなたは具体的な型の代わりに型パラメータを使って 1 つの方程式でそのような関数の全ての族を実装することができます。任意の型から Unit 型への多相的関数を何と呼ぶべきか？もちろん **unit** と呼びます：

```
unit :: a -> ()  
unit _ = ()  
  
def unit[A]: A => Unit = _ => ()
```

C++ では、この関数を次のように書きます:

```
template<class T>  
void unit(T) {}
```

次に、型の分類の中で二要素集合があります。C++ では **bool** と呼ばれ、Haskell では予測どおりに **Bool** と呼ばれます。違いは、C++ の **bool** は組み込み型であるのに対し、Haskell では次のように定義できます:

```
data Bool = True | False  
  
sealed trait Bool  
case object True extends Bool  
case object False extends Bool
```

(この定義の読み方は、**Bool** は **True** か **False** のどちらかであるということです。) 原理的には、C++ でも列挙型としてブール型を定義することができます:

```
enum bool {
    true,
    false
};
```

しかし、C++ の `enum` は実は整数です。C++11 の “`enum class`” を代わりに使用することができますが、その場合はその値をクラス名で修飾する必要があります。つまり、`bool::true` や `bool::false` といった形式で、それを使用するすべてのファイルに適切なヘッダを含める必要があります。

`Bool` からの純粋関数はターゲット型から 2 つの値を選びます。`True` に対応する 1 つの値と `False` に対応するもう 1 つの値です。

`Bool` への関数は述語 (*predicate*) と呼ばれます。例えば、Haskell ライブラリ `Data.Char` は `isAlpha` や `isDigit` のような述語でいっぱいです。C++ には `isalpha` や `isdigit` を定義する同様のライブラリがありますが、これらはブール型ではなく `int` を返します。実際の述語は `std::ctype` で定義され、`ctype::is(alpha, c)` や `ctype::is(digit, c)` などの形式を取ります。

2.7 チャレンジ

- お気に入りの言語で高階関数 (または関数オブジェクト) `memoize` (和訳: メモ化) を定義してください。この関数は純粋関数 `f` を引数として取り、`f` とほとんど同じように振る舞う

関数を返しますが、元の関数を各引数に対して一度だけ呼び出し、結果を内部に保存し、その後同じ引数で呼び出されるたびにこの保存された結果を返します。メモ化された関数は、そのパフォーマンスを見ることで元の関数と区別することができます。例えば、評価に長い時間がかかる関数をメモ化してみてください。最初にそれを呼び出したときは結果を待たなければなりませんが、同じ引数による後の呼び出しでは、結果が即座に得られるはずです。

2. 標準ライブラリのランダムな数を生成するために通常使用する関数をメモ化してみてください。うまくいきますか？
3. ほとんどのランダム数生成器はシードで初期化することができます。シードを取り、そのシードでランダム数生成器を呼び出し、結果を返す関数を実装してください。その関数をメモ化してください。うまくいきますか？
4. 以下の C++ 関数のどれが純粋ですか？ それらをメモ化し、メモ化されたものと、そうでないものそれぞれについて複数回呼び出したときに何が起こるかを観察してください。
 - (a) 本文中の例の階乗関数
 - (b) `std::getchar()`
 - (c) `bool f() {
 std::cout << "Hello!" << std::endl;
 return true;
}`
 - (d) `int f(int x) {`

```
static int y = 0;  
y += x;  
return y;  
}
```

5. **Bool** から **Bool** への異なる関数はいくつありますか？ それらをすべて実装できますか？
6. 対象が型 **Void**、**() (Unit)**、**Bool** のみである圏の図を描いてください。射はこれらの型間のすべての可能な関数に対応します。そして関数の名前で射をラベル付けしてください。

3

圏の大小

圏 を学ぶことで、様々な例を通して実際に圏についての深い理解が得られます。圏はあらゆる形や大きさで存在し、思いがけない場所に現れることがあります。まずは非常に単純なものから始めましょう。

3.1 対象なし

最も単純な圏は、対象も射もゼロの圏です。それ自体では非常に寂しい圏ですが、他の圏の文脈では重要かもしれません。例えば、全ての圏の圏(実際に存在します)の文脈では。空集合が意味を持つなら、空の圏が意味を持たない理由はありません。

3.2 単純なグラフ

対象を射で結びつけることにより、圏を構成することができます。任意の有向グラフから始めて、それを圏にするためにさらに射を追加していくことを想像してみてください。まず、各ノードに恒等射を追加します。次に、一方の終わりが他方の始まりと一致する(つまり、合成可能な)任意の2つの射について、それらの合成として機能する新しい射を追加します。新しい射を追加するたびに、他の射(恒等射を除く)および自身との合成を考慮する必要があります。通常、無限に多くの射になりますが、それは問題ありません。

このプロセスを別の方法で見ると、あなたはグラフの各ノードに対応する対象を持ち、合成可能なグラフのエッジのすべての可能なチェーンを射として持つ圏を作成していると言えます。(恒等射も長さゼロのチェーンとして考慮することもできます。)

このような圏は、与えられたグラフによって生成される自由圏と呼ばれます。これは自由構成の例であり、与えられた構造を最小限のアイテムで拡張してその規則(ここでは圏の規則)を満たすプロセスです。将来的には、それについてさらに多くの例を見るでしょう。

3.3 順序

そして今、完全に異なるものです！ 射が対象間の関係、すなわち小さいか等しい関係である圏について考えましょう。それが実際に圏で

あるかどうかを確認してみましょう。恒等射はありますか？すべての対象は自身に対して小さいか等しいです：チェック！ 合成はありますか？もし $a \leq b$ かつ $b \leq c$ ならば $a \leq c$ ：チェック！ 合成は結合的ですか？チェック！ このような関係を持つ集合は前順序と呼ばれ、前順序は確かに圈です。

また、もし $a \leq b$ かつ $b \leq a$ ならば $a = b$ と同じでなければならぬという追加条件を満たす、より強い関係もあります。それは半順序と呼ばれます。

最後に、任意の 2 つの対象が何らかの方法で互いに関連しているという条件を課すことができます。それによって得られるのは線形順序または全順序です。

これらの順序付けられた集合を圈として特徴付けましょう。前順序は、任意の対象 a から任意の対象 b への射が高々 1 つだけ存在する圈です。このような圈の別名は「痩せた圈」です。前順序は痩せた圈です。

圈 C における対象 a から対象 b への射の集合はホム集合と呼ばれ、 $C(a, b)$ （または、時には $\mathbf{Hom}_C(a, b)$ ）と表記されます。したがって、前順序のすべてのホム集合は空か単集合のいずれかです。それには対象 a から a への射の集合であるホム集合 $C(a, a)$ も含まれますが、これは任意の前順序で单一要素、つまり恒等射のみを含む必要があります。しかし、前順序にはサイクルが存在する可能性があります。サイクルは半順序では禁止されています。

前順序、半順序、および全順序を認識できることは非常に重要です。なぜなら、それらはソートに関連しているからです。クイックソート、バブルソート、マージソートなどのソートアルゴリズムは、全順序でのみ正しく機能します。半順序はトポロジカルソートを使用してソートできます。

3.4 モノイドとしての集合

モノイドは恥ずかしく単純ですが、驚くほど強力な概念です。それは基本的な算術の背後にある概念です: 加算と乗算の両方がモノイドを形成します。モノイドはプログラミングにおいて至る所に存在します。文字列、リスト、折りたたみ可能なデータ構造、並行プログラミングのフューチャー、関数型リアクティブプログラミングのイベントなどとして現れます。

伝統的に、モノイドは二項演算を持つ集合として定義されます。この演算に要求されるのは、それが結合的であり、それに関して振る舞いのような特別な要素があることだけです。

例えば、ゼロを含む自然数は、加算の下でモノイドを形成します。結合性とは、次のようなことを意味します:

$$(a + b) + c = a + (b + c)$$

(つまり、数を加算するときには括弧を省略できます。)

中立要素 (neutral element、単位要素) はゼロです。なぜなら:

$$0 + a = a$$

そして

$$a + 0 = a$$

2番目の方程式は冗長です。なぜなら加算は可換 $a + b = b + a$ ですが、可換性はモノイドの定義の一部ではありません。例えば、文字列の連結は可換ではありませんが、それでもモノイドを形成します。ちなみに、文字列の連結における中立要素は空文字列であり、文字列のどちらの側にも付け加えても変わりません。

Haskell ではモノイドのための型クラスを定義できます — それは中立要素を `mempty` と呼び、二項演算を `mappend` と呼ぶ型です:

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m

trait Monoid[M] {
    def combine(m1: M, m2: M): M
    def empty: M
}
```

2引数関数の型シグネチャ、 $m \rightarrow m \rightarrow m$ は最初は奇妙に見えるかもしれません、Curry化について話し合った後には完全に理解できるでしょう。複数の矢印を持つシグネチャを 2つの基本的な方法で解釈

することができます: 複数の引数を持つ関数として、右端の型が戻り値であるか、あるいは1つの引数(左端のもの)を持つ関数として、関数を返すものとしてです。後者の解釈は、矢印が右結合であるために冗長である括弧を追加することで強調されることがあります: $m \rightarrow (m \rightarrow m)$ 。この解釈については後ほど戻ってきます。

Haskell では、`mempty` と `mappend` のモノイダルな性質(つまり、`mempty` が中立であり、`mappend` が結合的であるという事実)を表現する方法がありません。それらが満たされていることを保証するのはプログラマの責任です。

Haskell のクラスは C++ のクラスほど侵襲的ではありません。新しい型を定義するとき、そのクラスを前もって指定する必要はありません。いくらでも先延ばしにして、後から任意の型をいくつかのクラスのインスタンスであると宣言する自由があります。例として、`String` を `mempty` と `mappend` の実装を提供することでモノイドとして宣言しましょう(これは標準の Prelude であなたのために行われます):

```
instance Monoid String where
    mempty = ""
    mappend = (++)

object Monoid {
    implicit val stringMonoid: Monoid[String] = new Monoid[String] {
        def combine(m1: String, m2: String): String = m1 + m2
        def empty: String = ""
    }
}
```

```
    }  
}
```

ここでは、**String** が文字のリストであるため、リストの連結演算子 (**++**) を再利用しています。

Haskell の構文について: どんな中置演算子も、括弧で囲むことにより 2 引数関数に変換することができます。2 つの文字列があれば、**++** を間に挿入することによりそれらを連結することができます:

```
"Hello" ++ "world!"
```

あるいは、括弧で囲んだ (**++**) に 2 つの引数としてそれらを渡すこともできます:

```
(++) "Hello" "world!"
```

関数への引数はコンマで区切られられず、括弧で囲まれません。(これは Haskell を学ぶ際に慣れるのが最も難しいことの一つかもしれません。)

Haskell が関数の等価性を表現することを許すことを強調することが価値があります。例えば、次のように表現します:

```
mappend = (++)
```

概念的には、これは関数が生成する値の等価性を表現することとは異なります。例えば:

```
mappend s1 s2 = (++) s1 s2
```

前者は圏 **Hask** (もしくは、bottoms、つまり永遠に終わらない計算を無視するならば **Set**) における射の等価性に翻訳されます。このような等式は、より簡潔であり、しばしば他の圏に一般化することができます。後者は拡張的等価性と呼ばれ、任意の 2 つの入力文字列に対して、**mappend** と **(++)** の出力が同じであるという事実を述べています。引数の値が時々ポイント (例: f の x での値) と呼ばれるため、これはポイントワイス等価性と呼ばれます。引数を指定せずに関数の等価性を述べることは、ポイントフリーとして記述されます。(ちなみに、ポイントフリー等式はしばしば関数の合成を含みますが、これは初学者にとっては少し混乱するかもしれません。)

C++ でモノイドを宣言する最も近い方法は、C++20 標準のコンセプト機能を使用することです。

```
template<class T>
struct mempty;

template<class T>
T mappend(T, T) = delete;

template<class M>
concept Monoid = requires (M m) {
    { mempty<M>::value() } -> std::same_as<M>;
    { mappend(m, m) } -> std::same_as<M>;
```

```
};
```

最初の定義は、各特殊化に対して中立要素を保持する構造です。

delete キーワードは、デフォルト値が定義されていないことを意味します: それはケースバイケースで指定される必要があります。同様に、**mappend** にはデフォルトがありません。

コンセプト **Monoid** は、与えられた型 **M** に対して適切な **mempty** と **mappend** の定義が存在するかどうかをテストします。

Monoid コンセプトのインスタンス化は、適切な特殊化とオーバーロードを提供することによって達成することができます:

```
template<>
struct mempty<std::string> {
    static std::string value() { return ""; }
};

template<>
std::string mappend(std::string s1, std::string s2) {
    return s1 + s2;
}
```

3.5 モノイドとしての圏

それは集合とその要素から離れて、対象と射について話すことを目指している圏論を知っているように、「二項演算を適用する」というこ

とは集合内で「動かす」または「シフトする」ことを意味すると少し考え方を変えてみましょう。

例えば、すべての自然数に 5 を加える操作があります。それは 0 を 5 に、1 を 6 に、2 を 7 にマッピングします。それは自然数の集合で定義された関数です。良いです：私たちは関数と集合を持っています。一般的に、任意の数 n に対して、 n を加える関数、「 n の加算関数」があります。

加算関数はどのように合成されますか？ 5 を加える関数と 7 を加える関数の合成は、12 を加える関数です。したがって、加算関数の合成は加算の規則と等価にすることができます。それも良いです：私たちは加算を関数の合成に置き換えることができます。

しかし、待ってください、もっとあります：中立要素、ゼロの加算関数もあります。ゼロを加えることは何も動かしません、それでそれは自然数の集合の恒等関数です。

私があなたに伝統的な加算の規則を与える代わりに、私は同じくらいの情報を失うことなく、加算関数の合成則を与えることができます。注意してください、加算関数の合成は関数の合成が結合的であるため、結合的です。そして私たちは、恒等関数に対応するゼロ加算関数を持っています。

気付くかもしれない識者は、整数から加算関数へのマッピングが `mappend` の型シグネチャの 2 番目の解釈、つまり

$m \rightarrow (m \rightarrow m)$ から従うと指摘するかもしれません。それは私たちに **mappend** がモノイド集合の要素をその集合上で作用する関数にマッピングすることを教えています。

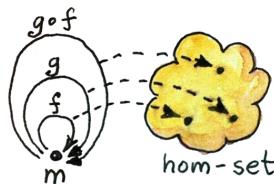
さて、あなたが自然数の集合を扱っているという事実を忘れて、それを单一の対象、射の束である塊として考えてみてください。モノイドは单一の対象の圏です。実際、モノイドという名前は、单一を意味するギリシャ語の *mono* から来ています。すべてのモノイドは、適切な合成則に従う射の集合を持つ单一の対象の圏として記述することができます。



文字列の連結は興味深いケースです。なぜなら、私たちは右アペンドナーと左アペンドナー（あるいはプレペンドナー、もしあなたが望むなら）を定義する選択肢を持っているからです。2つのモデルの合成テーブルは互いに逆転します。あなたは簡単に自分自身を納得させることができます。

できます: 「foo」の後に「bar」を追加することは、「bar」をプレpendした後に「foo」をprependすることに相当します。

あなたは、すべての圏論的モノイド – 単一対象の圏 – が一意の二項演算を備えた集合のモノイドを定義するかどうかという質問をするかもしれません。実は、单一対象の圏から常に集合を抽出することができます。この集合は、私たちの例の加算関数である射の集合です。言い換えれば、圏 \mathbf{M} の单一対象 m のホム集合 $\mathbf{M}(m, m)$ を持っています。私たちはこの集合内で二項演算を簡単に定義することができます: 2つの集合要素のモノイダル積は、対応する射の合成に対応する要素です。2つの $\mathbf{M}(m, m)$ の要素 f と g を与えられれば、それらの積は合成 $f \circ g$ に対応します。合成は常に存在します。なぜなら、これらの射のソースとターゲットは同じ対象だからです。そしてそれは圏の規則によって結合的です。恒等射はこの積の中立要素です。したがって、私たちは常に圏モノイドから集合モノイドを復元することができます。すべての意図と目的において、彼らは同一です。



モノイドホム集合が射として、そして集合内の点として見られる。

数学者が摘むべきちょっとした難点があります: 射が必ずしも集合を形成する必要はありません。圏の世界には、集合よりも大きいものが存在します。任意の 2 つの対象間の射が集合を形成する圏は、局所的に小さいと呼ばれます。私は約束したように、そのような微妙さをほとんど無視するつもりですが、記録に残しておくべきだと思いました。

圏論における多くの興味深い現象の根底には、ホム集合の要素を、合成則に従う射として、または集合内の点として見ることができるという事実があります。ここでは、 M の射の合成が集合 $M(m, m)$ 内のモノイダル積に変換されます。

3.6 チャレンジ

1. 以下から自由圏を生成してみましょう:
 - (a) ノードが 1 つでエッジがないグラフ
 - (b) ノードが 1 つで 1 つの(有向)エッジがあるグラフ(ヒント: このエッジは自分自身と合成できます)
 - (c) 2 つのノードとそれらの間に单一の矢印があるグラフ
 - (d) 単一のノードとアルファベットの文字でマークされた 26 の矢印があるグラフ: a, b, c ... z.
2. これはどのような種類の順序ですか?
 - (a) 含まれる関係による集合の集合: 集合 A が集合 B に含まれるのは、 A のすべての要素が B の要素でもある場合です。

- (b) 次のサブタイピング関係による C++ 型: T_1 が T_2 のサブタイプである場合、 T_1 へのポインターを T_2 へのポインターを期待する関数に渡すことができ、コンパイルエラーを引き起こさない。
3. **Bool** が **True** と **False** の 2 つの値の集合であることを考慮して、それがそれぞれ演算子 **&&** (AND) と **||** (OR) に関して 2 つの (集合論的) モノイドを形成することを示してください。
 4. AND 演算子を持つ **Bool** モノイドを圈として表現します: 射とその合成則をリストアップしてください。
 5. 3 を法とする加算をモノイド圏として表現します。

4

Kleisli 圈

型と純粋関数を圏としてモデル化する方法を見てきました。また、非純粋関数や副作用を圏論でモデル化することにも触れました。実行中にログやトレースを生成する関数の例を見てみましょう。このようなことは、命令型言語では、おそらく何らかのグローバル状態を変更することによって実装されるでしょう。例えば:

```
string logger;

bool negate(bool b) {
    logger += "Not so! ";
    return !b;
}
```

この関数が純粋ではないことは、メモ化されたバージョンがログを生成できないためです。この関数には副作用があります。

現代のプログラミングでは、可能な限りグローバルな可変状態から遠ざかるようにします – 特に並行処理の複雑さのためです。そして、このようなコードをライブラリに入れることはありません。

幸いなことに、この関数を純粋にすることは可能です。ログを明示的に渡すだけです。引数に文字列を追加し、通常の出力と更新されたログを含む文字列とをペアにしましょう:

```
pair<bool, string> negate(bool b, string logger) {
    return make_pair(!b, logger + "Not so! ");
}
```

この関数は純粋で、副作用がなく、同じ引数で呼ばれるたびに同じペアを返し、必要に応じてメモ化できます。しかし、ログの累積的な性質を考えると、与えられた呼び出しに至るまでのすべての可能な履歴をメモ化する必要があります。別々のメモエントリが必要になります:

```
negate(true, "It was the best of times. ");
```

そして

```
negate(true, "It was the worst of times. ");
```

などがあります。

これはまた、ライブラリ関数のインターフェースとしてあまり良くありません。呼び出し元は戻り値の文字列を無視することができますが、入力として文字列を渡すことを強制されます。これは不便かもしれません。

もっと侵入的でない方法で同じことをする方法はありませんか？関心の分離はできませんか？この単純な例では、関数 `negate` の主な目的は、あるブール値を別のブール値に変換することです。ログは二次的なものです。確かに、記録されるメッセージは関数に特有のものですが、メッセージを一つの連続したログに集約するタスクは別の問題です。私たちはまだ関数が文字列を生成することを望んでいますが、ログの生成からはそれを解放したいと思います。そこで妥協案です：

```
pair<bool, string> negate(bool b) {
    return make_pair(!b, "Not so! ");
}
```

アイディアは、ログが関数呼び出しの間に集約されるというものです。

これをどのように実行するかを見るために、もう少し現実的な例に切り替えましょう。文字列から文字列への関数が一つあり、小文字を大文字に変換します：

```
string toUpper(string s) {
    string result;
    int (*toupperp)(int) = &toupper; // toupper is overloaded
    transform(begin(s), end(s), back_inserter(result), toupperp);
```

```
    return result;  
}
```

そして、文字列を空白の境界で文字列のベクトルに分割する別の関数があります:

```
vector<string> toWords(string s) {  
    return words(s);  
}
```

実際の作業は補助関数 `words` で行われます:

```
vector<string> words(string s) {  
    vector<string> result{""};  
    for (auto i = begin(s); i != end(s); ++i)  
    {  
        if (isspace(*i))  
            result.push_back("");  
        else  
            result.back() += *i;  
    }  
    return result;  
}
```

私たちは関数 `toUpper` と `toWords` を変更して、それらが通常の戻り値の上にメッセージ文字列を載せるようにしたいと思っています。



これらの関数の戻り値を「飾る」ために、任意の型 `A` の値と文字列のペアをカプセル化するテンプレート `Writer` を定義することによって、一般的な方法で行います:

```
template<class A>
using Writer = pair<A, string>;
```

ここに飾られた関数があります:

```
Writer<string> toUpper(string s) {
    string result;
    int (*toupperp)(int) = &toupper;
    transform(begin(s), end(s), back_inserter(result), toupperp);
    return make_pair(result, "toUpper ");
}

Writer<vector<string>> toWords(string s) {
    return make_pair(words(s), "toWords ");
}
```

これらの二つの関数を、文字列を大文字に変換し、単語に分割し、その行動のログを生成する別の飾られた関数に合成したいと思っています。それを行う方法はこうです:

```
Writer<vector<string>> process(string s) {
    auto p1 = toUpper(s);
    auto p2 = toWords(p1.first);
    return make_pair(p2.first, p1.second + p2.second);
}
```

目標を達成しました: ログの集約はもはや個々の関数の懸念事項ではありません。それらは自分自身のメッセージを生成し、それから外部的に大きなログに連結されます。

こうして書かれた全プログラムを想像してみてください。これは繰り返し、間違いややすいコードの悪夢です。しかし、私たちはプログラマーです。繰り返しコードをどう扱うか知っています: それを抽象化します! ただし、これはごく普通の抽象化ではありません — 関数合成自体を抽象化する必要があります。しかし、合成は圏論の本質ですので、もっとコードを書く前に、問題を圏論的な視点から分析しましょう。

4.1 Writer 圈

関数の戻り値を飾ることで、何らかの追加機能を背負わせるというアイデアは非常に有益です。これに関する多くの例を見ることになる

でしょう。出発点は私たちの通常の型と関数の圏です。型を対象として残しますが、飾られた関数をもつ射に再定義します。

例えば、`int` から `bool` への関数 `isEven` を飾りたいとします。それを、ペアを返す飾られた関数で表現される射に変えます。重要な点は、この射がまだ `int` と `bool` の対象間の射と見なされることですが、飾られた関数がペアを返します:

```
pair<bool, string> isEven(int n) {
    return make_pair(n % 2 == 0, "isEven ");
}
```

圏の規則によれば、私たちはこの射を、`bool` の対象から何らかのものへの別の射と合成することができるはずです。特に、以前の `negate` と合成することができるはずです:

```
pair<bool, string> negate(bool b) {
    return make_pair(!b, "Not so! ");
}
```

明らかに、これら二つの射を通常の関数のように合成することはできません。なぜなら、入出力の不一致のためです。その合成は次のようになるはずです:

```
pair<bool, string> isOdd(int n) {
    pair<bool, string> p1 = isEven(n);
    pair<bool, string> p2 = negate(p1.first);
```

```
    return make_pair(p2.first, p1.second + p2.second);
}
```

そこで、この新しい圏での二つの射の合成のレシピです:

1. 最初の射に対応する飾られた関数を実行する
2. 結果ペアの最初のコンポーネントを抽出し、それを第二の射に
対応する飾られた関数に渡す
3. 最初の結果の二番目のコンポーネント(文字列)と、第二の結果
の二番目のコンポーネント(文字列)を連結する
4. 最終結果の最初のコンポーネントと連結された文字列を組み合
わせた新しいペアを返す

C++でこの合成を高階関数として抽象化したい場合、私たちは、私たちの圏の3つの対象に対応する3つの型でパラメータ化されたテンプレートを使わなければなりません。それは、私たちの規則に従って合成可能な二つの飾られた関数を取り、第三の飾られた関数を返します:

```
template<class A, class B, class C>
function<Writer<C>(A)> compose(function<Writer<B>(A)> m1,
                                         function<Writer<C>(B)> m2)
{
    return [m1, m2](A x) {
        auto p1 = m1(x);
        auto p2 = m2(p1.first);
```

```
    return make_pair(p2.first, p1.second + p2.second);
};

}
```

これで、以前の例に戻って、この新しいテンプレートを使って `toUpper` と `toWords` の合成を実装できます。

```
Writer<vector<string>> process(string s) {
    return compose<string, string, vector<string>>(toUpper, toWords)(s);
}
```

型の渡しに関するノイズはまだ多いです。しかし、一般化されたラムダ関数と戻り型の推論をサポートしている C++14 準拠のコンパイラを使用していれば、これを回避できます（このコードのクレジットは Eric Niebler にあります）：

```
auto const compose = [](auto m1, auto m2) {
    return [m1, m2](auto x) {
        auto p1 = m1(x);
        auto p2 = m2(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
    };
};
```

この新しい定義では、`process` の実装が簡単になります：

```
Writer<vector<string>> process(string s) {
    return compose(toUpper, toWords)(s);
}
```

しかし、まだ終わっていません。私たちは新しい圏での合成を定義しましたが、恒等射は何でしょうか？これらは通常の恒等関数ではありません！それらは型 A から型 A へ戻る射であり、それは以下の形式の飾られた関数です：

```
Writer<A> identity(A);
```

それらは合成に関して単位要素として振る舞う必要があります。合成の定義を見ると、恒等射は引数を変更せずに渡し、ログに空文字列を追加するだけでなければならないことがわかります：

```
template<class A> Writer<A> identity(A x) {
    return make_pair(x, "");
}
```

この新しく定義された圏が確かに正当な圏であることを簡単に確認できます。特に、私たちの合成は自明に結合的です。各ペアの最初のコンポーネントで何が起こっているかを追えば、それはただの通常の関数合成であり、それは結合的です。二番目のコンポーネントは連結されており、連結もまた結合的です。

洞察力のある読者は、この構造を文字列モノイドだけでなく、任意のモノイドに一般化するのは簡単だと気づくかもしれません。`compose`

内で `mappend` を使用し、`identity` 内で `mempty` を使用します (+ と "" の代わりに)。ログに文字列のみを使用することに制限する理由は実際にはありません。良いライブラリの作者は、ライブラリが機能するために必要な最小限の制約を特定できるはずです — ここでは、ログライブラリの唯一の要件は、ログがモノイダルな特性を持つことです。

4.2 Haskell での Writer

Haskell での同じことは少し簡潔であり、コンパイラから多くの助けを得られます。まず `Writer` 型を定義しましょう:

```
type Writer a = (a, String)  
  
type Writer[A] = (A, String)
```

ここで私は単なる型エイリアスを定義しています。これは C++ の `typedef` (または `using`) と同等です。`Writer` 型は型変数 `a` によってパラメータ化されており、`a` と `String` のペアに相当します。ペアの構文は最小限です: ただ二つのアイテムをカンマで区切って括弧に入れるだけです。

私たちの射は任意の型からある `Writer` 型への関数です:

```
a -> Writer b
```

```
A => Writer[B]
```

合成は時々「魚」と呼ばれる面白い中置演算子として宣言します:

```
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
```

```
def >=>[A, B, C](m1: A => Writer[B], m2: B => Writer[C]): A => Writer[C]
```

これはそれぞれが独自の関数である二つの引数の関数です。そして、関数を返します。最初の引数は `(a -> Writer b)` 型で、二番目は `(b -> Writer c)` 型であり、結果は `(a -> Writer c)` 型です。

ここにその中置演算子の定義があります — 二つの引数 `m1` と `m2` は魚のシンボルの両側に現れます:

```
m1 >=> m2 = \x ->
  let (y, s1) = m1 x
    (z, s2) = m2 y
  in (z, s1 ++ s2)
```

```
object kleisli {
  //allows us to use >=> as an infix operator
  implicit class KleisliOps[A, B](m1: A => Writer[B]) {
    def >=>[C](m2: B => Writer[C]): A => Writer[C] =
      x => {
        val (y, s1) = m1(x)
        val (z, s2) = m2(y)
```

```
        (z, s1 + s2)
    }
}
}
```

結果は一つの引数 `x` のラムダ関数です。ラムダはバックスラッシュで書かれています — これは、足を切断されたギリシャ文字の λ と考えてください。

`let` 式を使うと、補助変数を宣言できます。ここでは、`m1` への呼び出しの結果が変数のペア `(y, s1)` にパターンマッチされ、最初のパターンからの引数 `y` での `m2` への呼び出しの結果が `(z, s2)` にマッチされます。

Haskell では、ペアに対するパターンマッチを行うことが一般的です。C++ でアクセッサを使用した場合のように行いません。それ以外には、二つの実装間にかなり直接的な対応関係があります。

`let` 式の全体的な値は、その `in` 節で指定されます。ここでは、それは `z` の最初のコンポーネントと二つの文字列の連結 `s1++s2` の二番目のコンポーネントのペアです。

私たちの圈のための恒等射も定義しますが、後で明らかになる理由から、それを `return` と呼びます。

```
return :: a -> Writer a
return x = (x, "")
```

```
def pure[A](x: A): Writer[A] = (x, "")
```

完全性のために、飾られた関数 `upCase` と `toWords` の Haskell バージョンを持ちましょう:

```
upCase :: String -> Writer String
upCase s = (map toUpper s, "upCase ")
```

```
toWords :: String -> Writer [String]
toWords s = (words s, "toWords ")
```

```
val upCase: String => Writer[String] =
  s => (s.toUpperCase, "upCase ")
```

```
val toWords: String => Writer[List[String]] =
  s => (s.split(' ').toList, "toWords ")
```

関数 `map` は C++ の `transform` に相当します。それは文字関数 `toUpperCase` を文字列 `s` に適用します。補助関数 `words` は標準の Prelude ライブライアリで定義されています。

最後に、二つの関数の合成は魚演算子の助けを借りて達成されます:

```
process :: String -> Writer [String]
process = upCase >=> toWords
```

```
val process: String => Writer[List[String]] = {  
    import kleisli._  
    upCase >=> toWords  
}
```

4.3 Kleisli 圈

私がこの圏をその場で発明したわけではないと推測したかもしれません。これは、モナドに基づくいわゆる Kleisli 圈の一例です。私たちはまだモナドについて議論する準備ができていませんが、それらが何を達成できるかの一端をお見せしたかったのです。私たちの限られた目的のために、Kleisli 圈は、基礎となるプログラミング言語の型を対象として持ります。型 A から型 B への射は、特定の装飾を使用して B から派生した型への関数です。各 Kleisli 圈は、そのような射の合成方法と、その合成に関する恒等射を定義します。(後に見るように、「装飾」という漠然とした用語は、圏内の自己関手の概念に対応します。)

この章で使用した特定のモナドは、`writer` モナドと呼ばれ、関数の実行をロギングまたはトレースするために使用されます。これはまた、純粋な計算に効果を埋め込むためのより一般的なメカニズムの例でもあります。以前に見たように、我々はプログラミング言語の型と関数を(通常のように底を無視して)集合の圏でモデル化することができます。ここでは、射が飾られた関数によって表され、それらの合成がただ一つの関数の出力を別の関数の入力に渡すことを行う、少

し異なる圏へとこのモデルを拡張しました。合成自体で遊ぶことができる自由度がもう一つあります。それはちょうど、副作用を使って命令型言語で伝統的に実装されているプログラムに単純な表示的意味論を与えることができる自由度です。

4.4 チャレンジ

引数のすべての可能な値に対して定義されていない関数は、部分関数と呼ばれます。それは数学的な意味での本当の関数ではないので、標準的な圏論的型には当てはまりません。しかし、それは **optional** という装飾された型を返す関数によって表現することができます:

```
template<class A> class optional {
    bool _isValid;
    A _value;
public:
    optional() : _isValid(false) {}
    optional(A v) : _isValid(true), _value(v) {}
    bool isValid() const { return _isValid; }
    A value() const { return _value; }
};
```

例えば、次のような **safe_root** 関数の実装があります:

```
optional<double> safe_root(double x) {
    if (x >= 0) return optional<double>{sqrt(x)};
    else return optional<double>{};
}
```

ここで、チャレンジです:

1. 部分関数のための Kleisli 圈を構成します (合成と恒等射を定義します)。
2. それがゼロでない場合にその引数の有効な逆数を返す、飾られた関数 `safe_reciprocal` を実装します。
3. 関数 `safe_root` と `safe_reciprocal` を合成して、可能であれば `sqrt(1/x)` を計算する `safe_root_reciprocal` を実装します。

5

積と余積

古代ギリシャの劇作家、エウリピデスはかつてこう言いました: 「すべての人は、よく交わる仲間によって定義される。」関係によって私たちは定義される。圏論においてこれが真実である場所はない。特定の対象を圏内で一つだけにしたい場合、他の対象(および自己)との関係のパターンを説明することによってのみこれを行うことができます。これらの関係は射によって定義されます。

圏論には、対象をそれらの関係に関して定義するための普遍構成と呼ばれる共通の構成法があります。これを行う一つの方法は、対象と射から構成された特定の形、あるパターンを選び、圏内でそのすべての発生を探すことです。それが十分に一般的なパターンであり、かつ

圏が大きい場合、たくさんのヒットがあるでしょう。問題は、それらのヒットの中で何らかのランク付けを行い、最も適したものを選ぶことです。

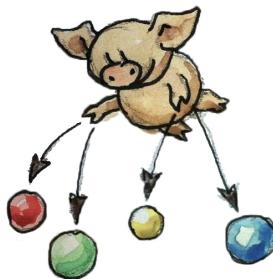
このプロセスは、私たちがウェブ検索を行う方法を思い起こさせます。クエリはある種のパターンのようなものです。非常に一般的なクエリは大きなリコールをもたらします：たくさんのヒットがあります。その中には関連するものもあればそうでないものもあります。関連しないヒットを排除するために、クエリを洗練します。それによってその精度を高めます。最終的に、検索エンジンはヒットをランク付けし、望む結果がリストのトップに来ることを期待します。

5.1 始対象

最も単純な形は单一の対象です。明らかに、この形のインスタンスは、与えられた圏の対象の数と同じです。選択するには多すぎます。何らかのランク付けを行い、この階層のトップにある対象を見つけると試みる必要があります。私たちの手元にある唯一の手段は射です。射を矢印と考えるなら、圏の一方の端から他方へと全体的な矢印の流れがあるかもしれません。これは、たとえば半順序のような順序圏で真実です。対象 a が対象 b よりも「より始まり側」であるということを、 a から b への矢印（射）がある場合に一般化することができます。そして、他のすべての対象に向かう矢印を持つ**始対象**を定義します。明らかにそのような対象が存在するとは限らないのですが、それで問

題ありません。より大きな問題は、そのような対象が多すぎるかもしれません。再現率 (recall) は良いが、適合率 (precision) に欠けます。解決策は、順序圏からヒントを得ることです。順序圏は、任意の 2 つの対象の間に高々 1 つの射しか許さない: 1 つの対象が他の対象よりも小さいか等しい一つの方法しかありません。これに導かれて始対象のこの定義に至ります:

始対象は、圏内の任意の対象に向かって 1 つだけ射を持つ対象です。



しかし、それでも始対象の一意性 (存在する場合) を保証するわけではありません。しかし、次善のことを保証します: 同型による一意性。同型は圏論において非常に重要であり、その理由については間もなく話します。今のところ、同型による一意性が始対象の定義に「the」を使用することを正当化することに同意しましょう。

いくつかの例を挙げましょう: 半順序集合(しばしば *poset* と呼ばれます)における始対象は、その最小要素です。いくつかの poset には始対象がありません。例えば、大小関係による射を持つすべての正と負の整数の集合です。

集合と関数の圏において、始対象は空集合です。空集合は Haskell の型 **Void** に対応し(C++ には対応する型がありません)、**Void** から他の任意の型への一意の多相的関数は **absurd** と呼ばれます:

```
absurd :: Void -> a
```

```
def absurd[A]: Nothing => A
```

Void を型の圏の始対象にするのは、この射の族です。

5.2 終対象

单一対象のパターンを続けてみましょうが、対象をランク付けする方法を変えます。対象 *a* が対象 *b* よりも「より終わり」であると言うことが、*b* から *a* への射がある場合に(矢印の方向が逆転したことに注意してください)。圏の中で他のどの対象よりも「より終わり」である対象を探します。再び、一意性を主張します:

終対象は、圏内の任意の対象からそれに向かって 1 つだけ射を持つ対象です。



そして再び、終対象は同型によって一意であり、それを間もなく示します。しかし、まずいくつかの例を見てみましょう。poset では、終対象が存在する場合、それは最大の対象です。集合の圏では、終対象は単集合です。私たちはすでに単集合について話しました。それは C++ の `void` 型と Haskell の `Unit` 型 `()` に対応しています。それは唯一の値を持つ型です。C++ では暗黙のうちに、Haskell では明示的に、`()` として示されます。また、任意の型から `Unit` 型への 1 つだけの純粹関数が存在することも確立しました:

```
unit :: a -> ()  
unit _ = ()  
  
def unit[A]: A => Unit = _ => ()
```

したがって、終対象のすべての条件が満たされています。

この例では、一意性の条件が重要であることに注意してください。なぜなら、空集合を除いて、他のすべての集合(実際にはそれらすべて)が各集合からの入ってくる射を持っているからです。例えば、次のようなブール値関数(述語)があります:

```
yes :: a -> Bool  
yes _ = True  
  
def yes[A]: A => Boolean = _ => true
```

しかし **Bool** は終対象ではありません。少なくとも **Void** 以外の各型からもう 1 つ以上の **Bool** 値関数があります:

```
no :: a -> Bool  
no _ = False  
  
def no[A]: A => Boolean = _ => false
```

一意性にこだわることで、終対象の定義を 1 つの型に正確に絞り込むことができます。

5.3 双対性

始対象と終対象を定義する方法の間には、明らかな対称性があります。唯一の違いは射の方向です。任意の圏 **C** に対して、すべての射を

逆転させることによって逆圏 C^{op} を定義することができます。逆圏は、同時に合成を再定義する限り、自動的に圏の要件を満たします。もともとの射 $f :: a \rightarrow b$ と $g :: b \rightarrow c$ が $h :: a \rightarrow c$ に合成され、 $h = g \circ f$ であった場合、逆転した射 $f^{op} :: b \rightarrow a$ と $g^{op} :: c \rightarrow b$ は $h^{op} :: c \rightarrow a$ に合成され、 $h^{op} = f^{op} \circ g^{op}$ になります。そして、恒等射を逆転させることは(冗談を言うと!)無操作です。

双対性は、圏の非常に重要な性質です。なぜなら、それは圏論に取り組むすべての数学者の生産性を倍増させるからです。あなたが考案したすべての構成物には、その反対があります。そして、あなたが証明したすべての定理には、無料で1つもらえます。逆圏の構成物はしばしば「余(co)」という接頭語が付けられるので、積と余積、モナドと余モナド、錐と余錐、極限と余極限などがあります。ただし、射を2回逆転させると元の状態に戻るため、余余モナドはありません。

それにより、終対象は逆圏における始対象であるということになります。

5.4 同型

プログラマとして、等価性を定義することが非自明な作業であることを私たちはよく知っています。2つの対象が等しいとはどういう意味ですか？彼らはメモリ内の同じ場所を占めなければならないのか？(ポインタの等価性) それとも、すべてのコンポーネントの値が等しいことで十分でしょうか？2つの複素数が実部と虚部で表される場合

と、大きさと角度で表される場合に等しいですか？ 数学者も等価性の意味を解明したわけではありません。彼らは提案的な等価性、内包的等価性、外延的等価性、ホモトピー型理論におけるパスとしての等価性など、複数の競合する等価性の定義を持っています。そして、それらは同型、さらに弱い概念である同値というより弱い概念を持っています。

直感的には、同型な対象は同じ形をしています。それぞれの対象の一部が一対一で他の対象の何らかの部分に対応しています。私たちの計測器具では、2つの対象は完璧なコピーです。数学的には、対象 a から対象 b への写像があり、対象 b から対象 a への写像があり、それらは互いに逆です。圏論では、写像を射に置き換えます。同型は、互いに逆である射のペア、または可逆な射です。

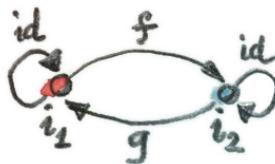
逆とは、合成と恒等射の観点で理解します。射 g が射 f の逆である場合、それらの合成は恒等射です。これは、2つの射を合成する2つの方法があるため、実際には2つの等式です：

```
f . g = id  
g . f = id
```

```
f compose g == identity _  
g compose f == identity _
```

始対象(終対象)が同型によって一意であると言ったとき、私は任意の2つの始対象(終対象)が同型であるという意味でした。それは実際に

は容易にわかります。2つの始対象 i_1 と i_2 を持っていると仮定します。 i_1 が始対象であるので、 i_1 から i_2 への一意の射 f があります。同様に、 i_2 が始対象であるので、 i_2 から i_1 への一意の射 g があります。これら2つの射の合成は何でしょうか？



この図のすべての射は一意です。

合成 $g \circ f$ は i_1 から i_1 への射でなければなりません。しかし、 i_1 は始対象なので、 i_1 から i_1 への射はただ一つしかありません。私たちは圏にいるので、 i_1 から i_1 への恒等射があることを知っています。そして、ただ一つの場所しかないので、それがそれでなければならない。したがって $g \circ f$ は恒等射と等しいです。同様に、 $f \circ g$ も恒等射と等しいです。なぜなら、 i_2 から i_2 へ戻る射はただ一つしかありません。これにより、 f と g は互いに逆でなければならないことが証明されます。したがって、任意の2つの始対象は同型です。

この証明では、始対象からそれ自身への射の一意性を使用しました。それがなければ、私たちは「同型による一意性」の部分を証明することができません。しかし、 f と g の一意性はなぜ必要なのでしょうか？それは、始対象が同型によって一意であるだけでなく、一意の同型に

よって一意であるからです。原理的には、2つの対象間に複数の同型が存在する可能性がありますが、ここではそうではありません。この「一意の同型による一意性」は、すべての普遍構成の重要な性質です。

5.5 積

次の普遍構成は、積です。2つの集合のデカルト積が何であるかはわかっています：それはペアの集合です。しかし、積集合とその構成要素をつなぐパターンは何でしょうか？それがわかれれば、他の圏にそれを一般化することができます。

積からそれぞれの構成要素への2つの関数、射影があるということだけが言えます。Haskellでは、これら2つの関数はそれぞれ `fst` と `snd` と呼ばれ、ペアの第一コンポーネントと第二コンポーネントを選びます：

```
fst :: (a, b) -> a
fst (x, y) = x
```

```
def fst[A, B]: ((A, B)) => A = {
    case (x, y) => x
}
```

```
snd :: (a, b) -> b
snd (x, y) = y
```

```
def snd[A, B]: ((A, B)) => B = {
    case (x, y) => y
}
```

ここでは、引数に対するパターンマッチングによって関数が定義されています: 任意のペアにマッチするパターンは (x, y) で、そのコンポーネントを変数 x と y に抽出します。

これらの定義は、ワイルドカードの使用によってさらに単純化することができます:

```
fst (x, _) = x
snd (_, y) = y

def fst[A, B]: ((A, B)) => A = _._1
def snd[A, B]: ((A, B)) => B = _._2
```

C++ では、たとえばテンプレート関数を使用します:

```
template<class A, class B> A
fst(pair<A, B> const & p) {
    return p.first;
}
```

この非常に限られた知識を武器に、集合の圏で 2 つの集合 a と b の積の構成につながる対象と射のパターンを定義しようとします。このパ

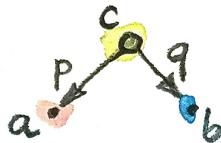
ターンは、対象 c と a と b にそれぞれ接続された 2 つの射 p と q で構成されます:

$p :: c \rightarrow a$

$q :: c \rightarrow b$

```
def p: C => A
```

```
def q: C => B
```



このパターンに適合するすべての c は、積の候補と見なされます。それらはたくさんあるかもしれません。



例えば、2 つの Haskell 型、**Int** と **Bool** を構成要素として選び、それらの積の候補のサンプリングを得ましょう。

こちらは一つ: **Int**。**Int** は **Int** と **Bool** の積の候補と見なすことができますか？ はい、それは可能です。そして、その射影はこちらです：

```
p :: Int -> Int
p x = x

q :: Int -> Bool
q _ = True

def p: Int => Int = x => x

def q: Int => Boolean = _ => true
```

それはかなり物足りないですが、基準に合っています。

こちらは別の例: (`Int, Int, Bool`)。それは 3 つの要素、または 3 つ組のタプルです。こちらはそれを正当な候補にする 2 つの射です (3 つ組に対するパターンマッチングを使用しています):

```
p :: (Int, Int, Bool) -> Int
p (x, _, _) = x

q :: (Int, Int, Bool) -> Bool
q (_, _, b) = b

def p: ((Int, Int, Boolean)) => Int = _._1

def q: ((Int, Int, Boolean)) => Boolean = _._3
```

お気づきかもしれません、最初の候補は小さすぎます。それは積の **Int** の次元だけをカバーしています。一方、2番目は大きすぎます。それは **Int** の次元を不必要に複製しています。

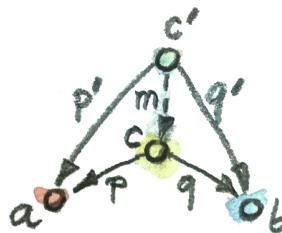
しかし、私たちはまだ普遍構成のもう一方の部分、ランキングを探検していません。私たちは2つのパターンのインスタンスを比較することができるようになります。候補の対象 c とその2つの射影 p と q を、別の候補の対象 c' とその2つの射影 p' と q' と比較したいです。 c が c' よりも「良い」と言いたいのですが、そのためには c' から c への射 m があることが不十分です。私たちは、その射影が c' の射影よりも「より普遍的」、「より良い」であることも望みます。それが意味するのは、射影 p' と q' が m を使って p と q から再構成できることです：

$$p' = p \circ m$$

$$q' = q \circ m$$

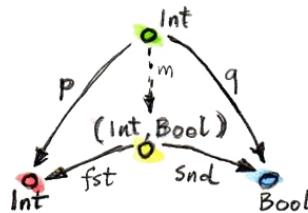
$$p1 == p \text{ compose } m$$

$$q1 == q \text{ compose } m$$



これらの等式を別の角度から見ると、 m が p' と q' を因数分解します。これらの等式が自然数であり、ドットが乗算であると想像してください。 m は p' と q' に共通する因子です。

少し直感を養うために、ペア (`Int`, `Bool`) と 2 つの標準的な射影、`fst` と `snd` が実際には前に提示した 2 つの候補よりも良いことを示しましょう。



最初の候補のための写像 `m` は:

```
m :: Int -> (Int, Bool)
m x = (x, True)

def m: Int => (Int, Boolean) = x => (x, true)
```

確かに、2 つの射影、`p` と `q` は再構成できます:

```
p x = fst (m x) = x
q x = snd (m x) = True
```

```
def p: Int => Int = x => fst(m(x)) // == x
def q: Int => Boolean = x => snd(m(x)) // == true
```

2番目の例のための `m` も同様に一意に決まります:

```
m (x, _, b) = (x, b)
```

```
def m: ((Int, Int, Boolean)) => (Int, Boolean) = {
    case (x, _, b) => (x, b)
}
```

私たちは `(Int, Bool)` が 2つの候補よりも良いことを示すことができました。逆が真実ではない理由を見てみましょう。`fst` と `snd` を `p` と `q` から再構成するための何らかの `m'` を見つけることができますか？

```
fst = p . m'
snd = q . m'
```

```
fst == p compose m1
snd == q compose m1
```

最初の例では、`q` は常に `True` を返し、私たちはペアの第二コンポーネントが `False` であるものがあることを知っています。`q` から `snd` を再構成することはできません。

第二の例は異なります: 私たちは `p` または `q` を実行した後でも十分な情報を保持していますが、`fst` と `snd` を因数分解する方法は複数あ

ります。というのも、`p` と `q` の両方が三つ組の第二コンポーネントを無視するため、`m'` はそれに何を入れても構いません。以下のようにすることができます:

```
m' (x, b) = (x, x, b)
```

```
def m1: ((Int, Boolean)) => (Int, Int, Boolean) = {  
    case (x, b) => (x, x, b)  
}
```

または

```
m' (x, b) = (x, 42, b)
```

```
def m1: ((Int, Boolean)) => (Int, Int, Boolean) = {  
    case (x, b) => (x, 42, b)  
}
```

などがあります。

すべてをまとめると、任意の型 `c` と 2 つの射影 `p` と `q` が与えられると、`c` からデカルト積 `(a, b)` への一意の `m` が存在し、それらを因数分解します。実際、それは単に `p` と `q` をペアに組み合わせるだけです。

```
m :: c -> (a, b)
```

```
m x = (p x, q x)
```

```
def m: C => (A, B) = x => (p(x), q(x))
```

これにより、デカルト積 (a, b) が最適な一致となり、これは集合の圏におけるこの普遍構成が機能していることを意味します。任意の 2 つの集合の積を選びます。

今、集合について忘れて、同じ普遍構成を使用して任意の圏で 2 つの対象の積を定義しましょう。そのような積が常に存在するわけではありませんが、存在するときは、一意の同型まで一意です。

2 つの対象 a と b の積は、2 つの射影を備えた対象 c であり、他の任意の対象 c' が 2 つの射影を備えている場合、それらの射影を因数分解する一意の射 m が c' から c へ存在します。

因数分解する関数を生成する(高次の)関数は、時々 **因数分解器** と呼ばれます。私たちの場合、それは以下の関数です:

```
factorizer :: (c -> a) -> (c -> b) -> (c -> (a, b))
factorizer p q = \x -> (p x, q x)
```

```
def factorizer[A, B, C]: (C -> A) => (C -> B) => (C -> (A, B)) =
  p => q => x => (p(x), q(x))
```

5.6 余積

圏論のすべての構成のように、積にも双対があり、余積と呼ばれます。積のパターンで射を逆転させると、2つの入射 (injection)、 i と j を備えた対象 c に終わります。これらは a と b から c への射です。

```
i :: a -> c
```

```
j :: b -> c
```

```
def i: A => C
```

```
def j: B => C
```



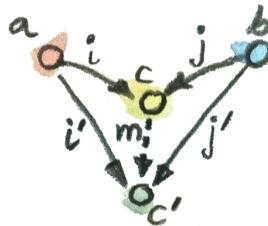
ランキングも逆転します: 対象 c が入射 i' と j' を備えた対象 c' よりも「良い」場合、 c から c' への射 m が存在し、その入射を因数分解します:

```
i' = m . i
```

```
j' = m . j
```

```
i1 == m compose i
```

```
j1 == m compose j
```



そのような対象の「最良」のもの、つまり他のどのパターンにも一意の射で結びつけるものは、余積と呼ばれ、存在する場合、一意の同型まで一意です。

2つの対象 a と b の余積は、2つの入射を備えた対象 c であり、他の任意の対象 c' が 2つの入射を備えている場合、それらの入射を因数分解する一意の射 m が c から c' へ存在します。

集合の圏では、余積は 2つの集合の非交和 (*disjoint union*) です。 a と b の非交和の要素は、 a の要素か b の要素のいずれかです。2つの集合が重なる場合、非交和には共通部分の 2つのコピーが含まれます。非交和の要素を、その起源を指定する識別子でタグ付けされていると考えることができます。

プログラマにとって、型の観点から余積を理解する方が簡単です。それは 2つの型のタグ付きユニオンです。C++ はユニオンをサポートしていますが、それらはタグ付けされていません。つまり、プログラムではどのユニオンメンバーが有効かどうかを何らかの方法で追跡す

る必要があります。タグ付きユニオンを作成するには、タグとしての列挙体を定義し、それをユニオンと組み合わせる必要があります。たとえば、`int` と `char const *` のタグ付きユニオンは、次のように実装することができます:

```
struct Contact {  
    enum { isPhone, isEmail } tag;  
    union { int phoneNum; char const * emailAddr; };  
};
```

2つの入射は、コンストラクタまたは関数として実装することができます。例えば、こちらは最初の入射としての関数 `PhoneNum` です:

```
Contact PhoneNum(int n) {  
    Contact c;  
    c.tag = isPhone;  
    c.phoneNum = n;  
    return c;  
}
```

これにより、整数が `Contact` に注入されます。

タグ付きユニオンは、バリエントとも呼ばれます。そして、boost ライブリには、`boost::variant` として非常に一般的なバリエントの実装があります。

Haskell では、任意のデータ型を縦棒で区切ったデータコンストラクタを使用してタグ付きユニオンに組み合わせることができます。

`Contact` の例は、宣言に次のように翻訳されます:

```
data Contact = PhoneNum Int | EmailAddr String

sealed trait Contact
case class PhoneNum(num: Int) extends Contact
case class EmailAddr(addr: String) extends Contact
```

ここで、`PhoneNum` と `EmailAddr` は、コンストラクタ(入射)としても機能し、パターンマッチングのためのタグとしても機能します(後で詳しく説明します)。たとえば、これは電話番号を使用してコンタクトを作成する方法です:

```
helpdesk :: Contact
helpdesk = PhoneNum 2222222

def helpdesk: Contact = PhoneNum(2222222)
```

プリミティブペアとして Haskell に組み込まれている積の標準的な実装とは異なり、余積の標準的な実装は `Either` というデータ型で、標準 Prelude で次のように定義されています:

```
data Either a b = Left a | Right b
```

```
sealed trait Either[A, B]
case class Left[A](v: A) extends Either[A, Nothing]
case class Right[B](v: B) extends Either[Nothing, B]
```

これは 2 つの型、`a` と `b` にパラメータ化されており、2 つのコンストラクタがあります: `a` 型の値を取る `Left` と、`b` 型の値を取る `Right` です。

積のために因数分解器を定義したように、余積のためにも定義することができます。候補の型 `c` と 2 つの候補の入射 `i` と `j` が与えられた場合、`Either` の因数分解器は因数分解関数を生成します:

```
factorizer :: (a -> c) -> (b -> c) -> Either a b -> c
factorizer i j (Left a) = i a
factorizer i j (Right b) = j b
```

```
def factorizer[A, B, C]: (A => C) => (B => C) => Either[A, B] => C =
  i => j => {
    case Left(a) => i(a)
    case Right(b) => j(b)
  }
```

5.7 非対称性

私たちは 2 つの双対な定義セットを見てきました: 始対象の定義は射の方向を逆転させることによって終対象の定義から得られ、同様に、余積の定義は積の定義から得られます。それでも、集合の圏では始対

象は終対象とは非常に異なり、余積は積と非常に異なります。後で見るように、積は乗法のように振る舞い、終対象は 1 の役割を果たし、一方で余積は和のように振る舞い、始対象は 0 の役割を果たします。特に有限集合の場合、積のサイズは個々の集合のサイズの積であり、余積のサイズはサイズの和です。

これは、射の反転に関して集合の圏が対称ではないことを示しています。

空集合は任意の集合に一意の射 (**absurd** 関数) を持っていますが、それに戻ってくる射はありません。一方、単集合は任意の集合から一意の射を持っていますが、それは(空集合を除いて)すべての集合に出ていく射も持っています。前に見たように、終対象からのこれらの出ていく射は、他の集合から要素を選ぶための非常に重要な役割を果たします(空集合には要素がないので、何も選ぶことはできません)。

単集合を Unit 型 () として表し、さらに別の候補として積のパターンに使用することで、それを他のものとは区別します。それに 2 つの射影 \mathbf{p} と \mathbf{q} を装備します。それぞれが単集合から構成要素の集合のそれぞれに向かう関数です。積が普遍的であるため、単集合から積への(一意の)射 \mathbf{m} もまたあります。この射は積集合から具体的なペアを選び

出し、また 2 つの射影を因数分解します:

```
p = fst . m  
q = snd . m
```

```
p == fst compose m  
q == snd compose m
```

単集合の唯一の要素 $\textcircled{0}$ に作用するとき、これら 2 つの等式はなります：

```
p \textcircled{0} = fst (m \textcircled{0})  
q \textcircled{0} = snd (m \textcircled{0})
```

```
p(\textcircled{0}) == fst(m(\textcircled{0}))  
q(\textcircled{0}) == snd(m(\textcircled{0}))
```

m $\textcircled{0}$ が m によって選ばれた積の要素であるので、これらの等式は p によって最初の集合から選ばれた要素、 p $\textcircled{0}$ がペアによって m によって選ばれたものの第一コンポーネントであることを教えてくれます。同様に、 q $\textcircled{0}$ は第二コンポーネントに等しいです。これは、積の要素が構成要素の集合の要素のペアであるという私たちの理解と完全に一致しています。

余積に対しては、そう単純な解釈はありません。単集合を余積の候補として試してみて、それから要素を抽出しようとするかもしれませんのが、そこでは 2 つの入射がそれに入るのあって、2 つの射影がそれから出てくるではありません。彼らは彼らの源について何も教えてくれません（実際、私たちはそれらが入力パラメータを無視することを見ました）。同様に、余積から私たちの単集合への一意の射もそうで

す。集合の圏は、始対象から見たときと終対象から見たときとで非常に異なるように見えます。

これは集合の本質的な性質ではなく、**Set**において射として使用される関数の性質です。一般に、関数は非対称です。させてください説明します。

関数は、その始域のすべての要素に対して定義されなければなりません（プログラミングでは、これを**全関数**と呼びます）が、その全域をカバーする必要はありません。私たちは、始域から単集合への関数—単集合の中でただ1つの要素を選ぶ関数—などの極端なケースを見てきました。（実際には、空集合からの関数が真の極端です。）始域のサイズが終域のサイズよりもはるかに小さい場合、私たちはしばしばそのような関数を、始域を終域に埋め込むと考えます。たとえば、単集合からの関数を使用して、その单一の要素を終域に埋め込むを考えることができます。私はそれらを**埋め込み関数**と呼びますが、数学者は反対を名付けることを好みます：その全域をしっかりと埋める関数は**全射**または**上への**と呼ばれます。

もう一つの非対称性の源は、関数が始域の多くの要素を終域の1つの要素にマッピングできるということです。それらはそれらを崩壊させることができます。極端なケースは、全集合を単集合にマッピングする関数です。あなたはそのような多相的な**unit**関数を見ました。崩壊は合成によってのみ複合することができます。2つの崩壊関数の合成は、個々の関数よりもさらに崩壊しています。数学者は非崩壊関数に名前を付けています：それらは**単射**または**一対一**と呼ばれます。

もちろん、埋め込みでも崩壊でもないいくつかの関数があります。それらは**全単射**と呼ばれ、それらは本当に対称的です。なぜなら、それらは可逆だからです。集合の圏では、同型は全単射と同じです。

5.8 チャレンジ

1. 終対象が一意の同型まで一意であることを示してください。
2. poset における 2 つの対象の積とは何ですか？ ヒント：普遍構成を使ってください。
3. poset における 2 つの対象の余積とは何ですか？
4. お気に入りの言語 (Haskell 以外) で Haskell の **Either** に相当する汎用型を実装してください。
5. **Either** が次の 2 つの入射を備えた **int** よりも「良い」余積であることを示してください：

```
int i(int n) { return n; }
int j(bool b) { return b ? 0: 1; }
```

ヒント：関数

```
int m(Either const & e);
```

を定義して、**i** と **j** を因数分解してください。

6. 前の問題を引き続き: どうやって `int` が 2 つの入射 `i` と `j` を備えていても `Either` より「良い」余積にはなり得ないと主張しますか？
7. まだ続けて: 以下の入射についてはどうでしょう？

```
int i(int n) {  
    if (n < 0) return n;  
    return n + 2;  
}  
  
int j(bool b) { return b ? 0: 1; }
```

8. `int` と `bool` の余積の劣った候補を考えてみてください。それが `Either` への複数の受け入れ可能な射を許容するため、それが `Either` より優れていることはありません。

5.9 参考文献

1. The Catsters, Products and Coproducts^{*1} video.

^{*1} <https://www.youtube.com/watch?v=upCSDIO9pjc>

6

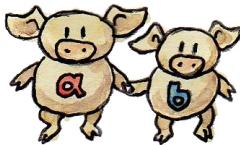
単純代数的データ型

型 を組み合わせる基本的な方法として、積と余積を見てきました。日常的なプログラミングでの多くのデータ構造は、これら二つの仕組みを使って構成できることがわかります。これは重要な実践的な意味を持ちます。データ構造の多くの特性は合成可能です。例えば、基本型の値の等価性を比較する方法と、これを積型や余積型に一般化する方法を知っていれば、複合型の等価演算子の導出を自動化できます。Haskell では等価性、比較、文字列への変換およびその逆変換などを、複合型の大部分に対して自動的に導出できます。

プログラミングにおける積型と和型をもう少し詳しく見てみましょう。

6.1 積型

プログラミング言語における二つの型の積の標準的な実装はペアです。Haskell ではペアはプリミティブな型コンストラクタです; C++ では標準ライブラリで定義された比較的複雑なテンプレートです。



ペアは厳密には可換ではありません: `(Int, Bool)` のペアは `(Bool, Int)` のペアに置き換えることはできませんが、同じ情報を持っています。しかし、同型性を考慮すれば可換です – この同型性は `swap` 関数(自身の逆関数である)によって与えられます:

```
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

```
def swap[A, B]: ((A, B)) => (B, A) = {
    case (x, y) => (y, x)
}
```

これらの二つのペアを、同じデータを異なる形式で格納していると考えることができます。これはビッグエンディアンとリトルエンディアンのようなものです。

任意の数の型を積に組み合わせることができますが、ペアをペアの中に入れ子にする方法よりも簡単な方法があります: 入れ子になったペアはタプルと等価です。これは異なる方法でペアを入れ子にすることが同型であるという事実の帰結です。例えば、`a`、`b`、そして `c` という三つの型をこの順番で積に組み合わせたい場合、二つの方法でこれを行うことができます:

`((a, b), c)`

`((A, B), C)`

または

`(a, (b, c))`

`(A, (B, C))`

これらの型は異なります – 一方を他方が期待する関数に渡すことはできません – しかし、その要素は一対一の対応を持ちます。一方を他方に写像する関数があります:

```
alpha :: ((a, b), c) -> (a, (b, c))
alpha ((x, y), z) = (x, (y, z))

def alpha[A, B, C]: (((A, B), C)) => ((A, (B, C))) = {
    case ((x, y), z) => (x, (y, z))
}
```

そして、この関数は可逆です:

```
alpha_inv :: (a, (b, c)) -> ((a, b), c)
alpha_inv (x, (y, z)) = ((x, y), z)

def alphaInv[A, B, C]: ((A, (B, C))) => (((A, B), C)) = {
    case (x, (y, z)) => ((x, y), z)
}
```

したがって、これは同型です。これらは単に同じデータを異なる方法で再パッケージングする異なる方法です。

型の積の生成を、型に対する二項演算として解釈することができます。その観点から、上記の同型はモノイドで見た結合則に非常に似ています:

$$(a * b) * c = a * (b * c)$$

ただし、モノイドの場合、二つの積の組み合わせ方は等価でしたが、ここでは「同型上で等しい」だけです。

同型を受け入れることができ、厳密な等価性にこだわらない場合、さらに進んで Unit 型 () を、1 が乗算の単位要素であるのと同じ方法で積の単位として示すことができます。確かに、ある型 a の値と単位のペアリングは、いかなる情報も追加しません。型:

```
(a, ())
```

```
(A, Unit)
```

は a と同型です。ここに同型があります:

```
rho :: (a, ()) -> a
rho (x, ()) = x
```

```
def rho[A]: ((A, Unit)) => A = {
  case (x, ()) => x
}
```

```
rho_inv :: a -> (a, ())
rho_inv x = (x, ())
```

```
def rhoInv[A]: A => (A, Unit) =
  x => (x, ())
```

これらの観察は、集合の圏 Set がモノイダル圏であると言うことによって形式化できます。これは、対象(ここではデカルト積を取る)を

乗算できる意味で、圏でありながらもモノイドです。モノイダル圏について、そして完全な定義については後で詳しく説明します。

特に、和型と組み合わせる場合、Haskell で積型を定義するより一般的な方法があります。これは複数の引数を持つ名前付きコンストラクタを使用します。例えば、ペアは次のように代替的に定義できます：

```
data Pair a b = P a b

sealed trait Pair[A, B]
case class P[A, B](a: A, b: B) extends Pair[A, B]
```

ここで、`Pair a b` は他の二つの型、`a` と `b` によってパラメータ化された型の名前であり、`P` はデータコンストラクタの名前です。二つの型を `Pair` 型コンストラクタに渡すことでペア型を定義します。適切な型の二つの値をコンストラクタ `P` に渡すことでペア値を構成します。例えば、`String` と `Bool` のペアとして `stmt` という値を定義しましょう：

```
stmt :: Pair String Bool
stmt = P "This statement is" False

val stmt: Pair[String, Boolean] =
  P("This statement is", false)
```

最初の行は型宣言です。これは、`Pair` の型コンストラクタを使用し、`a` と `b` を `Pair` の汎用定義で置き換える `String` と `Bool` を使用します。

二行目は、データコンストラクタ `P` に具体的な文字列と具体的な布尔値を渡すことによって実際の値を定義します。型コンストラクタは型を構成するために使用され、データコンストラクタは値を構成するために使用されます。

型とデータコンストラクタの名前空間が Haskell で分かれているため、両方に同じ名前が使われることがよくあります、例えば：

```
data Pair a b = Pair a b

case class Pair[A, B](a: A, b: B)
```

そして十分に目を細めると、組み込みのペア型もこの種の宣言の変形であると見なすことができます。ここでは名前 `Pair` が二項演算子 `(,)` に置き換えられています。実際には、`(,)` を他の名前付きコンストラクタのように使用し、接頭辞記法を使用してペアを作成することができます：

```
stmt = (,) "This statement is" False

val stmt = ("This statement is", false)
```

同様に、`(,,)` を使用して三つ組を作成し、それ以降も同様です。

汎用ペアやタプルの代わりに、特定の名前付き積型を定義することもできます、例えば：

```
data Stmt = Stmt String Bool  
  
case class Stmt(s: String, b: Boolean)
```

これは単に `String` と `Bool` の積ですが、それ自体の名前とコンストラクタが与えられています。この宣言スタイルの利点は、同じ内容でも異なる意味と機能を持ち、互いに置換できない多くの型を定義できることです。

タプルや複数引数コンストラクタを使用したプログラミングは、どのコンポーネントが何を表しているかを追跡することが雑然としており、エラーが発生しやすいものです。コンポーネントに名前を付けることがしばしば望ましいです。名前付きフィールドを持つ積型は、Haskell でレコードと呼ばれ、C で `struct` と呼ばれます。

6.2 レコード

簡単な例を見てみましょう。私たちは、二つの文字列、名前と記号; そして整数、原子番号; を一つのデータ構造に結合することで化学元素を記述したいと思います。タプル (`String`, `String`, `Int`) を使用して、どのコンポーネントが何を表しているかを覚えておくことができます。コンポーネントはパターンマッチングによって抽出され、次のような関数で行われます。この関数は、元素の記号がその名前の接

頭辞であるかどうかをチェックします(例えば、**He** が **Helium** の接頭辞であるように):

```
startsWithSymbol :: (String, String, Int) -> Bool
startsWithSymbol (name, symbol, _) = isPrefixOf symbol name

val startsWithSymbol: ((String, String, Int)) => Boolean = {
    case (name, symbol, _) => name.startsWith(symbol)
}
```

このコードはエラーが発生しやすく、読みにくく、メンテナンスが困難です。レコードを定義する方がずっと良いでしょう:

```
data Element = Element { name :: String
                           , symbol :: String
                           , atomicNumber :: Int }

case class Element(
    name: String,
    symbol: String,
    atomicNumber: Int
)
```

二つの表現は、以下の二つの変換関数によって証明されるように、互いに同型です。これらはお互いの逆です:

```
tupleToElem :: (String, String, Int) -> Element
tupleToElem (n, s, a) = Element { name = n
                                , symbol = s
                                , atomicNumber = a }

val tupleToElem: ((String, String, Int)) => Element = {
  case (n, s, a) => Element(n, s, a)
}

elemToTuple :: Element -> (String, String, Int)
elemToTuple e = (name e, symbol e, atomicNumber e)

val elemToTuple: Element => (String, String, Int) =
  e => (e.name, e.symbol, e.atomicNumber)
```

レコードフィールドの名前は、これらのフィールドにアクセスする関数としても機能します。たとえば、`atomicNumber e` は `e` から `atomicNumber` フィールドを取り出します。`atomicNumber` を関数として使用すると、型:

```
atomicNumber :: Element -> Int

val atomicNumber: Element => Int
```

`Element` のレコード構文を使うと、関数 `startsWithSymbol` はより読みやすくなります:

```
startsWithSymbol :: Element -> Bool  
startsWithSymbol e = isPrefixOf (symbol e) (name e)
```

```
val startsWithSymbol: Element -> Boolean =  
  e => e.name startsWith e.symbol
```

Haskell のトリックを使って、関数 `isPrefixOf` を逆引用符で囲んで中置演算子に変え、ほとんど文のように読むことができるようになります:

```
startsWithSymbol e = symbol e `isPrefixOf` name e
```

```
val startsWithSymbol: Element -> Boolean =  
  e => e.name startsWith e.symbol
```

この場合、括弧は省略できます。なぜなら、中置演算子は関数呼び出しよりも優先順位が低いからです。

6.3 和型

集合の圏における積が積型を生じるように、余積も和型を生じます。 Haskell で和型の典型的な実装は次のようになります:

```
data Either a b = Left a | Right b
```

```
sealed trait Either[+A, +B]
case class Left[A](v: A) extends Either[A, Nothing]
case class Right[B](v: B) extends Either[Nothing, B]
```

そして、ペアと同様に、`Either` も可換です(同型性まで)し、入れ子にすることができる、入れ子の順序は関係ありません(同型性まで)。例えば、三つ組の和の等価物を定義することができます:

```
data OneOfThree a b c = Sinistral a | Medial b | Dextral c

sealed trait OneOfThree[+A, +B, +C]
case class Sinistral[A](v: A) extends OneOfThree[A, Nothing, Nothing]
case class Medial[B](v: B) extends OneOfThree[Nothing, B, Nothing]
case class Dextral[C](v: C) extends OneOfThree[Nothing, Nothing, C]
```

等々。

実際、集合の圏 `Set` は余積に関しても(対称的な)モノイダル圏です。二項演算の役割は不連続和によって演じられ、単位要素の役割は始対象によって演じられます。型の観点では、`Either` がモノイダル演算子として、`Void`、すなわち住人のいない型がその中立要素として機能します。`Either` をプラスと考え、`Void` をゼロと考えることができます。実際に、`Void` を和型に加えてもその内容は変わりません。例えば:

```
Either a Void
```

```
Either[A, Nothing]
```

は `a` と同型です。それはこの型の `Right` バージョンを構成する方法がないためです — `Void` の値は存在しません。`Either a Void` の唯一の住人は `Left` コンストラクタを使って構成され、単に `a` 型の値をカプセル化しています。したがって、象徴的に $a + 0 = a$ です。

和型は Haskell ではかなり一般的ですが、その C++ の等価物である `union` や `variant` はあまり一般的ではありません。その理由はいくつかあります。

まず、最も単純な和型は単なる列挙型であり、C++ では `enum` を使って実装されます。Haskell の和型の等価物:

```
data Color = Red | Green | Blue
```

```
sealed trait Color
case object Red extends Color
case object Green extends Color
case object Blue extends Color
```

の C++ 版は次のようになります:

```
enum { Red, Green, Blue };
```

さらに単純な和型:

```
data Bool = True | False

sealed trait Bool
case object True extends Bool
case object False extends Bool
```

は C++ のプリミティブな `bool` です。

値の存在または不在をエンコードする単純な和型は、空の文字列、負の数、`null` ポインタなどの特別なトリックや「不可能な」値を使って C++ でさまざまに実装されています。この種のオプショナリティは、意図的であれば、Haskell では `Maybe` 型を使って表現されます：

```
data Maybe a = Nothing | Just a

sealed trait Option[+A]
case object None extends Option[Nothing]
case class Some[A](a: A) extends Option[A]
```

`Maybe` 型は二つの型の和です。二つのコンストラクタを個別の型に分離すると、最初のものは次のようにになります：

```
data NothingType = Nothing

case object NoneType
```

これは **Nothing** と呼ばれる値を一つ持つ列挙型です。言い換えると、それは単集合であり、Unit 型 () と同等です。二番目の部分:

```
data JustType a = Just a

case class SomeType[A](a: A)
```

は単に型 a をカプセル化しています。私たちは、**Maybe** を次のようにエンコードできたでしょう:

```
type Maybe a = Either () a

type Option[A] = Either[Unit, A]
```

より複雑な和型は、しばしば C++ でポインタを使用して偽装されます。ポインタは null であるか、または特定の型の値を指しているかのどちらかです。例えば、(再帰的な) 和型として定義できる Haskell のリスト型:

```
data List a = Nil | Cons a (List a)

sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[A](h: A, t: List[A]) extends List[A]
```

は、空リストを実装するための null ポインタのトリックを使用して C++ に翻訳することができます:

```
template<class A>
class List {
    Node<A> * _head;
public:
    List() : _head(nullptr) {} // Nil
    List(A a, List<A> l)      // Cons
        : _head(new Node<A>(a, l))
    {}
};


```

二つの Haskell コンストラクタ **Nil** と **Cons** は、同様の引数 (**Nil** にはなし、**Cons** には値とリスト) を持つ二つのオーバーロードされた **List** コンストラクタに翻訳されます。和型の二つのコンポーネントを区別するためにタグを必要としない代わりに、**_head** の特別な **nullptr** 値を使用して **Nil** をエンコードします。

しかし、Haskell と C++ の型の主な違いは、Haskell のデータ構造が不变であることです。特定のコンストラクタを使ってオブジェクトを作成すると、そのオブジェクトは永遠にどのコンストラクタが使われたか、どの引数がそれに渡されたかを覚えています。したがって、**Just "energy"** として作成された **Maybe** オブジェクトは決して **Nothing** にはなりません。同様に、空のリストは永遠に空であり、3 つの要素を持つリストは常に同じ 3 つの要素を持ちます。

この不変性が構成を可逆にします。オブジェクトが与えられたら、それを構成に使用されたパートまで常に分解することができます。この分解はパターンマッチングで行われ、コンストラクタはパターンとして再利用されます。コンストラクタの引数、もしあれば、変数(または他のパターン)に置き換えられます。

`List` データ型には二つのコンストラクタがあるため、任意の `List` の分解は、それらのコンストラクタに対応する二つのパターンを使います。一つは空の `Nil` リストにマッチし、もう一つは `Cons` で構成されたリストにマッチします。例えば、ここに `List` 上の単純な関数の定義があります:

```
maybeTail :: List a -> Maybe (List a)
maybeTail Nil = Nothing
maybeTail (Cons _ t) = Just t

def maybeTail[A]: List[A] => Option[List[A]] = {
    case Nil => None
    case Cons(_, t) => Some(t)
}
```

`maybeTail` の定義の最初の部分は、パターンとして `Nil` コンストラクタを使用し、`Nothing` を返します。二番目の部分は、`Cons` コンストラクタをパターンとして使用します。最初のコンストラクタ引数をワイルドカードで置き換えます。なぜなら、私たちはそれに興味がないからです。`Cons` への二番目の引数は変数 `t` に束縛されます(これらのも

のを変数と呼びますが、厳密に言えば、それらは決して変化しません：一度式に束縛されると、変数は決して変わりません）。返り値は `Just t` です。今、あなたの `List` がどのように作成されたかに応じて、それは一つの節にマッチします。それが `Cons` を使って作成された場合、それに渡された二つの引数が回収されます（そして最初のものが破棄されます）。

さらに複雑な和型は、C++ で多型クラス階層を使用して実装されます。共通の祖先を持つクラスの族は、一つの変種型として理解することができます。その中で、`vtable` は隠されたタグとして機能します。Haskell ではコンストラクタにパターンマッチングを行い、特殊なコードを呼び出すことが行われるのと同様のことが、C++ では `vtable` ポイントに基づいて仮想関数への呼び出しをディスパッチすることで達成されます。

`union` が C++ で和型としてあまり使われない理由は、そこに入れることができるものに重大な制限があるためです。コピー構造体を持つ `std::string` を `union` に入れることさえできません。

6.4 型の代数

個別に取り扱うと、積型と和型は多様な有用なデータ構造を定義するためには使用することができますが、2つを組み合わせることから真の強みが生まれます。再び、構成の力を呼び出しています。

これまでに発見したことを要約してみましょう。我々は、**Void** を中立要素とする和型と、**Unit** 型 () を中立要素とする積型という、2つの可換モノイダル構造を型システムの下に見てきました。それらを加算と乗算に類似させて考えたいと思います。この類推では、**Void** はゼロに、単位、() は一に相当するでしょう。

この類推をどこまで伸ばすことができるか見てみましょう。例えば、ゼロによる乗算はゼロを与えるのか？ 言い換えれば、一つのコンポーネントが **Void** である積型は **Void** と同型ですか？ 例えば、**Int** と **Void** のペアを作成することは可能でしょうか？

ペアを作成するためには2つの値が必要です。整数を簡単に思いつくことはできますが、**Void** 型の値はありません。したがって、任意の型 **a** に対して、型 (**a**, **Void**) は住人がいない – 値がない – したがって **Void** と同等です。言い換えれば、 $a \times 0 = 0$ です。

加算と乗算を結び付けるもう一つのことは分配則です：

$$a \times (b + c) = a \times b + a \times c$$

これも積型と和型のために成り立ちますか？ はい、それは成り立ちます – いつものように、通常は同型性までです。左辺に対応する型は：

(**a**, **Either** **b** **c**)

(**A**, **Either**[**B**, **C**])

そして右辺に対応する型は：

```
Either (a, b) (a, c)
```

```
Either[(A, B), (A, C)]
```

一方向への変換を行う関数はこちらです:

```
prodToSum :: (a, Either b c) -> Either (a, b) (a, c)
```

```
prodToSum (x, e) =
```

```
  case e of
```

```
    Left y -> Left (x, y)
```

```
    Right z -> Right (x, z)
```

```
def prodToSum[A, B, C]: ((A, Either[B, C])) => Either[(A, B), (A, C)] = {
```

```
  case (x, e) => e match {
```

```
    case Left(y) => Left((x, y))
```

```
    case Right(z) => Right((x, z))
```

```
}
```

```
}
```

そして、もう一方への変換を行う関数はこちらです:

```
sumToProd :: Either (a, b) (a, c) -> (a, Either b c)
```

```
sumToProd e =
```

```
  case e of
```

```
    Left (x, y) -> (x, Left y)
```

```
    Right (x, z) -> (x, Right z)
```

```
def sumToProd[A, B, C]: Either[(A, B), (A, C)] => (A, Either[B, C]) = {  
    case Left((x, y)) => (x, Left(y))  
    case Right((x, z)) => (x, Right(z))  
}
```

`case of` 文は関数内でのパターンマッチングに使用されます。各パターンは矢印に続いて、パターンがマッチした時に評価される式が続きます。例えば、値を持って `prodToSum` を呼び出す場合:

```
prod1 :: (Int, Either String Float)  
prod1 = (2, Left "Hi!")
```

```
val prod1: (Int, Either[String, Float]) =  
(2, Left("Hi!"))
```

`e` は `case e of` で `Left "Hi!"` と等しくなります。それはパターン `Left y` と一致し、"Hi!" を `y` として置き換えます。既に `x` に `2` と一致させているため、`case of` 節の結果、そして関数全体の結果は、期待通り `Left (2, "Hi!")` になります。

これら二つの関数が互いに逆であることを証明するつもりはありませんが、それらについて考えると、そうでなければならぬとわかります！ それらはただ二つのデータ構造の内容を単純に再パッケージングしているだけです。同じデータ、ただ異なるフォーマットです。

これらののような互いに結びついたモノイドには数学者が名前をつけています：それを 半環 と呼びます。それは完全な環ではありません、な

ぜなら我々は型の減算を定義することができないからです。それがなぜ半環は時々 *rig*、つまり「n (negative、否定的) なしの環」と呼ばれるのかという理由です。しかしながら、それを除いて、我々は自然数についての声明、それが *rig* を形成するという声明を型についての声明に翻訳することから多くの利益を得ることができます。ここに興味のあるいくつかのエントリーの翻訳表があります:

数字	型
0	<code>Void</code>
1	<code>()</code>
$a + b$	<code>Either a b = Left a Right b</code>
$a \times b$	<code>(a, b)</code> または <code>Pair a b = Pair a b</code>
$2 = 1 + 1$	<code>data Bool = True False</code>
$1 + a$	<code>data Maybe = Nothing Just a</code>

リスト型はかなり興味深いです、なぜならそれは方程式の解として定義されているからです。我々が定義している型は方程式の両側に現れます:

```
data List a = Nil | Cons a (List a)
```

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[A](h: A, t: List[A]) extends List[A]
```

もし我々が通常の置換を行い、また `List a` を `x` と置き換えると、我々は方程式を得ます:

$$x = 1 + a * x$$

我々は伝統的な代数的手法を使用してそれを解くことはできません、なぜなら我々は型を減算したり除算したりすることができないからです。しかし、我々は右辺の `x` を `(1 + a*x)` と置き換え、分配則を使って続ける一連の置換を試みることができます。これは以下のシリーズにつながります:

$$x = 1 + a*x$$

$$x = 1 + a*(1 + a*x) = 1 + a + a*a*x$$

$$x = 1 + a + a*a*(1 + a*x) = 1 + a + a*a + a*a*a*x$$

...

$$x = 1 + a + a*a + a*a*a + a*a*a*a\dots$$

我々は無限の積(タブル)の和で終わります、それは次のように解釈されます: リストは空、`1` であるか、单一要素、`a` であるか、ペア、`a*a` であるか、三つ組、`a*a*a` であるか、等々…まさにリストがそうです—`a` の列！

リストにはそれ以上のものがあり、我々は関手と不動点について学んだ後にリストと他の再帰的データ構造に戻ってきます。

記号変数を使った方程式の解法、それが代数です！ それがこれらの型に彼らの名前を与えるものです: 代数的データ型。

最後に、型の代数の非常に重要な解釈について言及するべきです。二つの型 **a** と **b** の積は **a** と **b** の両方の値を含まなければならないことに注意してください、これは両方の型が居住されていなければならぬことを意味します。一方で二つの型の和は、**a** または **b** の値を含むので、それらの一つが居住されていれば十分です。*and* と *or* の論理も半環を形成し、それも型理論にマッピングすることができます:

論理	型
<i>false</i>	Void
<i>true</i>	()
<i>a b</i>	Either a b = Left a Right b
<i>a && b</i>	(a, b)

この類似はより深く、論理と型理論の間の Curry-Howard 対応の基礎を形成しています。我々は関数型について話すときにそれに戻ってきます。

6.5 チャレンジ

1. **Maybe a** と **Either () a** の間の同型を示してください。
2. Haskell で定義された和型をこちらです:

```
data Shape = Circle Float
           | Rect Float Float
```

`Shape` に対して行動するような関数、例えば `area` を定義したいとき、我々は二つのコンストラクタに対してパターンマッチングを行います:

```
area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rect d h) = d * h
```

C++ または Java で `Shape` をインターフェースとして実装し、`Circle` と `Rect` の二つのクラスを作成してください。`area` を仮想関数として実装してください。

3. 前の例に引き続き: `Shape` の周囲を計算する `circ` という新しい関数を簡単に追加できます。我々は `Shape` の定義に触れることなくそれを行うことができます:

```
circ :: Shape -> Float
circ (Circle r) = 2.0 * pi * r
circ (Rect d h) = 2.0 * (d + h)
```

あなたの C++ または Java の実装に `circ` を追加してください。元のコードのどの部分に触りましたか？

4. さらに進んで: 新しい形状、`Square` を `Shape` に追加し、必要なすべての更新を行ってください。Haskell と C++ または Java では

どのコードに触れなければならなかったか？(あなたが Haskell プログラマでなくても、変更はかなり明白であるはずです。)

5. 同型の違いを除いて (up to isomorphism)、型に対して $a+a = 2 \times a$ が成り立つことを示してください。私たちの翻訳表によれば、2 が **Bool** に対応することを思い出してください。

7

関手

壊れたレコードのように聞こえることを覚悟して、関手について言っておきます: 関手は非常にシンプルでありながら強力なアイディアです。圏論はそうしたシンプルで強力なアイディアでいっぱいです。関手とは圏間の写像です。2つの圏、 C と D を与えられたとき、関手 F は C の対象を D の対象へ写像します — これは対象に対する関数です。もし a が C の対象なら、私たちはその D での像を Fa (括弧なし) と書きます。しかし、圏は単に対象だけではありません — それは対象とそれらを結ぶ射です。関手はまた射を写像します — これは射に対する関数です。しかし、それは射を無造作に写像するのではありません — それは接続を保存します。従って、もし射 f が C で対象 a

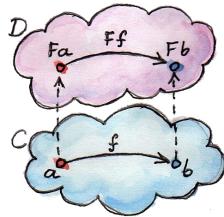
から対象 b へと繋がっているならば、

$$f :: a \rightarrow b$$

D での f の像、 Ff は a の像を b の像へと繋ぐでしょう：

$$Ff :: Fa \rightarrow Fb$$

(これは数学と Haskell の表記法を混合したもので、今までに理解していただけたと願います。関手を対象や射に適用する際、私は括弧を使いません。)

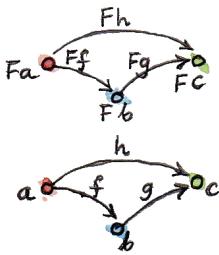


ご覧のとおり、関手は圏の構造を保存します：一つの圏で繋がっているものは、他の圏でも繋がっています。しかし、圏の構造にはもう一つ重要なことがあります：射の合成です。もし h が f と g の合成ならば：

$$h = g \circ f$$

私たちは F によるその像が f と g の像の合成であることを望みます：

$$Fh = Fg \circ Ff$$



最終的に、私たちは \mathbf{C} の全ての恒等射が \mathbf{D} の恒等射へ写像されることを望みます:

$$F\mathbf{id}_a = \mathbf{id}_{Fa}$$

ここで、 \mathbf{id}_a は対象 a での恒等射であり、 \mathbf{id}_{Fa} は Fa での恒等射です。

$$F\mathbf{id}_a = \mathbf{id}_{Fa}$$

$$\mathbf{id}_a$$

これらの条件は関手を通常の関数よりもずっと制限的にします。関手は圏の構造を保存しなければなりません。もし圏を対象が射によって結び付けられたコレクションとして想像するならば、関手はこの布にいかなる裂け目も導入してはいけません。それは対象をまとめたり、

複数の射を一つにまとめることはできますが、決して物事を分解してはなりません。この引き裂きを許さない制約は、あなたが微積分から知っているかもしれない連続性の条件に似ています。この意味で関手は「連続的」です(ただし、関手に対してもっと制限的な連続性の概念が存在します)。関数のように、関手は崩壊させることも埋め込むこともできます。ソース圏がターゲット圏よりもずっと小さい場合、埋め込む側面がより目立ちます。極端な場合、ソースは自明な单一対象圏になります——一つの対象と一つの射(恒等射)を持つ圏です。单一対象圏から他の任意の圏への関手は、単にその圏の中の対象を一つ選びます。これは完全に、单集合からの射がターゲット集合の要素を選ぶ性質と類似しています。最大限に崩壊させる関手は定値関手 Δ_c と呼ばれます。それはソース圏の全ての対象をターゲット圏の選ばれた一つの対象 c に写像します。また、ソース圏の全ての射をターゲット圏の恒等射 id_c に写像します。それはブラックホールのように、全てを一つの特異点に凝縮させます。極限や余極限を議論する際に、この関手についてもっと見ることになるでしょう。

7.1 プログラミングにおける関手

地に足をつけてプログラミングについて話しましょう。私たちは型と関数の圏を持っています。私たちは自己関手と呼ばれる、この圏を自分自身へ写像する関手について話すことができます。では、型の圏における自己関手とは何でしょうか？ まず第一に、それは型を型に写

像します。私たちはそのような写像の例を、それがまさにそれであると気づかずに見てきました。私が話しているのは、他の型によってパラメータ化された型の定義です。いくつかの例を見てみましょう。

7.1.1 Maybe 関手

Maybe の定義は型 `a` から型 `Maybe a` への写像です:

```
data Maybe a = Nothing | Just a

sealed trait Option[+A]
case object None extends Option[Nothing]
case class Some[A](a: A) extends Option[A]
```

ここで重要な繊細さがあります: `Maybe` 自体は型ではなく、**型コンストラクタ**です。あなたはそれに型引数を与えなければなりません、例えば `Int` や `Bool` のように、それを型に変えるために。引数なしの `Maybe` は型に対する関数を表します。しかし、私たちは `Maybe` を関手にすることができるでしょうか？(今後、プログラミングの文脈で関手について話すとき、私はほぼ常に自己関手を意味します。) 関手は対象(ここでは型)の写像だけでなく、射(ここでは関数)の写像でもあります。任意の関数 `a` から `b` へ:

```
f :: a -> b
```

```
val f: A => B
```

私たちは `Maybe a` から `Maybe b` への関数を生成したいと思います。このような関数を定義するためには、`Maybe` の 2 つのコンストラクタに対応する 2 つのケースを考慮する必要があります。`Nothing` のケースは単純です：私たちは単に `Nothing` を返します。そしてもし引数が `Just` ならば、私たちは関数 `f` をその内容に適用します。従って `f` の `Maybe` の下での像は次の関数です：

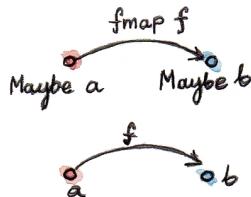
```
f' :: Maybe a -> Maybe b
f' Nothing = Nothing
f' (Just x) = Just (f x)
```

```
def f1[A, B]: Option[A] => Option[B] = {
  case None => None
  case Some(x) => Some(f(x))
}
```

(ところで、Haskell では変数名にアポストロフィを使うことができます。これはこのような場合に非常に便利です。) Haskell では、関手の射写像部分を `fmap` と呼ばれる高階関数として実装します。`Maybe` の場合、それは次のようなシグネチャを持ちます：

```
fmap :: (a -> b) -> (Maybe a -> Maybe b)
```

```
def fmap[A, B](f: A => B): (Option[A] => Option[B])
```



私たちはしばしば **fmap** が関数を **持ち上げる**と言います。持ち上げられた関数は **Maybe** 値に作用します。通常の Curry 化のために、このシグネチャは 2 つの方法で解釈されるかもしれません: 関数 $(a \rightarrow b)$ を一つの引数として持つ関数として – これは関数 $(\text{Maybe } a \rightarrow \text{Maybe } b)$ を返します; または 2 つの引数を持つ関数として **Maybe b** を返します:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
def fmap[A, B](f: A => B)(fa: Option[A]): Option[B]
```

私たちの前の議論に基づいて、これが **Maybe** に対する **fmap** の実装です:

```
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)
```

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
    case None => None
    case Some(x) => Some(f(x))
}
```

型コンストラクタ `Maybe` と関数 `fmap` が関手を形成することを示すためには、`fmap` が恒等性と合成を保存することを証明しなければなりません。これらは「関手則」と呼ばれます、それらは単に圏の構造の保存を保証するだけです。

7.1.2 等式推論

関手則を証明するために、私は等式推論を使います。これは Haskell で一般的な証明手法です。それは Haskell 関数が等式として定義される事実を利用します：左辺は右辺と等しいです。あなたは常に一方を他方に代入することができます、変数の名前の衝突を避けるために名前を変更することが必要かもしれません。これを関数のインライソング、あるいは逆に、式を関数にリファクタリングすると考えてください。例えば、次のような恒等関数を考えてみましょう：

```
id x = x

def identity[A](x: A) = x
```

例えば、ある式で `id y` を見た場合、あなたはそれを `y` と置き換えることができます（INLINE化）。さらに、式に `id` が適用されている場合、例えば `id (y + 2)` を見た場合、それを式自体 `(y + 2)` で置き換えることができます。そして、この置換は双方向で機能します：あなたは任意の式 `e` を `id e` で置き換えることができます（リファクタリング）。関数がパターンマッチングによって定義されている場合、あなたはそれぞれのサブ定義を独立して使用することができます。例えば、上記の `fmap` の定義を与えられた場合、`fmap f Nothing` を `Nothing` と置き換えることができます、またはその逆も可能です。これが実際にどのように機能するか見てみましょう。恒等性の保存から始めましょう：

```
fmap id = id  
  
fmap(identity) == identity
```

`Nothing` と `Just` の 2 つのケースを考慮する必要があります。こちらが最初のケースです（私は Haskell の疑似コードを使用して左辺を右辺に変換します）：

```
fmap id Nothing  
= { definition of fmap }  
  Nothing  
= { definition of id }  
  id Nothing
```

最後のステップで `id` の定義を逆向きに使用しました。式 `Nothing` を `id Nothing` と置き換えるしました。実際には、このような証明を行う際には、「両端からキャンドルを燃やす」ようにして、中央で同じ式に達するまで進めます — ここではそれが `Nothing` でした。2 番目のケースも同様に簡単です：

```
fmap id (Just x)
= { definition of fmap }
  Just (id x)
= { definition of id }
  Just x
= { definition of id }
  id (Just x)
```

次に、`fmap` が合成を保存することを示しましょう：

```
fmap (g . f) = fmap g . fmap f
fmap(g compose f) == fmap(g) compose fmap(f)
```

まず `Nothing` のケースから：

```
fmap (g . f) Nothing
= { definition of fmap }
  Nothing
= { definition of fmap }
  fmap g Nothing
```

```
= { definition of fmap }
fmap g (fmap f Nothing)
```

それから `Just` のケース:

```
fmap (g . f) (Just x)
= { definition of fmap }
Just ((g . f) x)
= { definition of composition }
Just (g (f x))
= { definition of fmap }
fmap g (Just (f x))
= { definition of fmap }
fmap g (fmap f (Just x))
= { definition of composition }
(fmap g . fmap f) (Just x)
```

等式推論が副作用を持つ C++ スタイルの「関数」には機能しないことを強調しておく価値があります。このコードを考えてみてください:

```
int square(int x) {
    return x * x;
}

int counter() {
    static int c = 0;
    return c++;
}
```

```
double y = square(counter());
```

等式推論を使えば、`square` をインライン化して以下のようになります：

```
double y = counter() * counter();
```

これは明らかに有効な変換ではなく、同じ結果を生み出すことはありません。それにもかかわらず、C++ コンパイラは `square` をマクロとして実装している場合、等式推論を試みるでしょう、その結果は災害になる可能性があります。

7.1.3 Optional

関手は Haskell で容易に表現されますが、ジェネリックプログラミングと高階関数をサポートするどんな言語でも定義することができます。C++ の `Maybe` の類似物であるテンプレート型 `optional` を考えてみましょう。こちらが実装のスケッチです (実際の実装は引数の渡され方、コピー意味論、および C++ 特有のリソース管理問題など、さまざまな側面を扱うため、はるかに複雑です) :

```
template<class T>
class optional {
    bool _isValid; // the tag
    T _v;
public:
```

```
optional()      : _isValid(false) {}           // Nothing
optional(T x) : _isValid(true) , _v(x) {} // Just
bool isValid() const { return _isValid; }
T val() const { return _v; } };
```

このテンプレートは関手の定義の一部を提供します: 型の写像です。それは任意の型 `T` を新しい型 `optional<T>` に写像します。その関数的作用を定義しましょう:

```
template<class A, class B>
std::function<optional<B>(optional<A>)>
fmap(std::function<B(A)> f) {
    return [f](optional<A> opt) {
        if (!opt.isValid())
            return optional<B>{};
        else
            return optional<B>{ f(opt.val()) };
    };
}
```

これは引数として関数を取

り、関数を返す高階関数です。こちらが Curry 化されていないバージョンです:

```
template<class A, class B>
optional<B> fmap(std::function<B(A)> f, optional<A> opt) {
    if (!opt.isValid())
        return optional<B>{};
```

```
    else
        return optional<B>{ f(opt.val()) };
}
```

`fmap` を `optional` のテンプレートメソッドとするオプションもあります。選択肢の多さが、C++ で関手パターンを抽象化する問題を生み出しています。関手は継承するインターフェースとして定義されるべきですか（残念ながら、仮想テンプレート関数を持つことはできません）？それは Curry 化された自由なテンプレート関数か、あるいは Curry 化されていない自由なテンプレート関数か？C++ コンパイラは欠けている型を正しく推測できるでしょうか、それともそれらは明示的に指定されるべきですか？入力関数 `f` が `int` から `bool` へのものである場合、コンパイラは `g` の型をどのようにして推測するでしょうか：

```
auto g = fmap(f);
```

特に将来的には、`fmap` をオーバーロードする複数の関手が存在する可能性がある場合です。（もうすぐ私たちはもっと多くの関手を見ることになります。）

7.1.4 型クラス

では、Haskell は関手の抽象化をどのように扱っているのでしょうか？それは型クラスメカニズムを使用します。型クラスは共通のイン

ターフェースをサポートする型の族を定義します。例えば、等価性をサポートするオブジェクトのクラスは次のように定義されます:

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

trait Eq[A]{
  def ===(x: A, y: A): Boolean
}
```

この定義は、型 `a` が 2 つの `a` 型の引数を取り、`Bool` を返す演算子 `(==)` をサポートする場合、`Eq` クラスの型であると述べています。特定の型が `Eq` であると Haskell に伝えたい場合、それをこのクラスのインスタンスと宣言し、`(==)` の実装を提供しなければなりません。例えば、2 つの `Float` の積型である 2 次元の `Point` の定義が与えられた場合:

```
data Point = Pt Float Float

case class Point(x: Float, y: Float)
```

点の等価性を定義することができます:

```
instance Eq Point where
  (Pt x y) == (Pt x' y') = x == x' && y == y'
```

```
implicit val pointEq = new Eq[Point] {  
    def ==(a1: Point, a2: Point): Boolean =  
        a1.x == a2.x && a1.y == a2.y  
}
```

ここで、演算子 (==) (私が定義しているもの) を 2 つのパターン (`Pt x y` と `(Pt x' y')`) の間の中置位置で使用しました。関数の本体は単一の等号に続いています。いったん `Point` が `Eq` のインスタンスとして宣言されると、直接点を等価性で比較することができます。C++ や Java とは異なり、`Point` を定義する際に `Eq` クラス (またはインターフェース) を指定する必要はなく、クライアントコードで後から行うことができます。型クラスはまた、関数 (および演算子) のオーバーロードのための Haskell 唯一のメカニズムです。私たちは異なる関手に対して `fmap` をオーバーロードするためにそれを必要とします。ただし、一つの複雑さがあります: 関手は型ではなく、型の写像、型コンストラクタとして定義されます。私たちは型ではなく、型コンストラクタの族である型クラスが必要です。幸いなことに、Haskell の型クラスは型コンストラクタと同じくらい型でも機能します。こちらが `Functor` クラスの定義です:

```
class Functor f where  
    fmap :: (a -> b) -> f a -> f b
```

```
trait Functor[F[_]] {  
    def fmap[A, B](f: A => B)(fa: F[A]): F[B]  
}
```

それは `f` が指定された型シグネチャを持つ関数 `fmap` が存在する場合、`Functor` であると規定します。小文字の `f` は型変数であり、型変数 `a` および `b` に似ています。しかし、コンパイラは、それが他の型に作用することによって、それが型コンストラクタであるという事実を推測することができます、例えば `f a` と `f b` のように。それに応じて、`Functor` のインスタンスを宣言する際には、`Maybe` のケースのように、型コンストラクタを与えなければなりません:

```
instance Functor Maybe where  
    fmap _ Nothing = Nothing  
    fmap f (Just x) = Just (f x)  
  
implicit val optionFunctor = new Functor[Option] {  
    def fmap[A, B](f: A => B)(fa: Option[A]): Option[B] = fa match {  
        case None => None  
        case Some(x) => Some(f(x))  
    }  
}
```

ちなみに、**Functor** クラスとそのインスタンス定義の多くは、**Maybe** を含む多くの単純データ型と共に標準の Prelude ライブラリの一部です。

7.1.5 C++ における関手

同じアプローチを C++ で試すことはできますか？ 型コンストラクタに相当するものはテンプレートクラス、例えば **optional** のようなものなので、類推により、**fmap** をテンプレートテンプレートパラメータ **F** でパラメータ化したいと思います。これがそのための構文です：

```
template<template<class> F, class A, class B>
F<B> fmap(std::function<B(A)>, F<A>);
```

私たちは異なる関手に対してこのテンプレートを特殊化することができるようにならなければなりませんが、残念ながら C++ ではテンプレート関数の部分特殊化が禁止されています。以下のように書くことはできません：

```
template<class A, class B>
optional<B> fmap<optional>(std::function<B(A)> f, optional<A> opt)
```

代わりに、関数のオーバーロードに頼らなければなりません。これにより、元の Curry 化されていない **fmap** の定義に戻ります：

```
template<class A, class B>
optional<B> fmap(std::function<B(A)> f, optional<A> opt) {
    if (!opt.isValid())
        return optional<B>{};
    else
        return optional<B>{ f(opt.val()) };
}
```

この定義は機能しますが、第二引数の `fmap` がオーバーロードを選択するため、より一般的な定義の `fmap` を完全に無視します。

7.1.6 リスト関手

プログラミングにおける関手の役割についての直感を得るために、さらに多くの例を見る必要があります。任意の型が別の型によってパラメータ化される場合、それは関手の候補です。ジェネリックコンテナは保存している要素の型によってパラメータ化されるため、非常にシンプルなコンテナ、リストを見てみましょう：

```
data List a = Nil | Cons a (List a)

sealed trait List[+E]
case object Nil extends List[Nothing]
case class Cons[E](h: E, t: List[E]) extends List[E]
```

私たちは型コンストラクタ `List` を持っており、それは任意の型 `a` を型 `List a` に写像します。`List` が関手であることを示すためには、関数の持ち上げを定義する必要があります: 関数 `a -> b` を与えられて、関数 `List a -> List b` を定義します:

```
fmap :: (a -> b) -> (List a -> List b)
```

```
def fmap[A, B](f: A => B): (List[A] => List[B])
```

`List a` に作用する関数は 2 つのケースを考慮する必要があります、それはリストの 2 つのコンストラクタに対応します。`Nil` のケースは単純です – 単に `Nil` を返します – 空のリストに対してはそれ以上することはありません。`Cons` のケースは少し複雑ですが、再帰を含みます。私たちが何をしようとしているのか少し立ち止まって考えてみましょう。私たちは `a` のリストを持っており、`a` を `b` に変換する関数 `f` を持っています。`f` を使用してリストの各要素を `a` から `b` に変換することが明らかです。では、実際にどのようにしてこれを行うのでしょうか？ 非空リストがヘッドとテールの `Cons` として定義されていることを考えると、ヘッドに `f` を適用し、テールにリフトされた (`fmap` された) `f` を適用します。これは再帰的な定義です。私たちはリフトされた `f` を、それを定義しているリストよりも短いリストに適用しています – それはそのテールです。私たちはより短いリストに向かって再帰していますので、最終的には空のリスト、つまり `Nil` に到達するはずです。しかし、

先に決定したように、`fmap f` が `Nil` に作用すると `Nil` を返します、これによって再帰が終了します。最終結果を得るために、新しいヘッド (`f x`) と新しいテール (`fmap f t`) を `Cons` コンストラクタを使用して組み合わせます。全てをまとめると、こちらがリスト関手のインスタンス宣言です：

```
instance Functor List where
    fmap _ Nil = Nil
    fmap f (Cons x t) = Cons (f x) (fmap f t)

implicit val listFunctor = new Functor[List] {
    def fmap[A, B](f: A => B)(fa: List[A]): List[B] = fa match {
        case Nil => Nil
        case Cons(x, t) => Cons(f(x), fmap(f)(t))
    }
}
```

C++ に慣れているなら、`std::vector` を考えてみてください。これは最も一般的な C++ コンテナであると考えられています。`std::vector` の `fmap` の実装は、`std::transform` の薄いカプセル化に過ぎません：

```
template<class A, class B>
std::vector<B> fmap(std::function<B(A)> f, std::vector<A> v) {
    std::vector<B> w;
    std::transform( std::begin(v)
```

```
        , std::end(v)
        , std::back_inserter(w)
        , f);
return w;
}
```

例えば、数列の要素を二乗するためにそれを使用することができます:

```
std::vector<int> v{ 1, 2, 3, 4 };
auto w = fmap([](int i) { return i*i; }, v);
std::copy( std::begin(w)
        , std::end(w)
        , std::ostream_iterator(std::cout, ", "));
```

`std::transform` に渡すことができるイテレータを実装しているほとんどの C++ コンテナは、よりプリミティブな `fmap` のいとこである `std::transform` によって、関手として機能します。残念ながら、イテレータや一時オブジェクトの通常のごちゃごちゃ (上記の `fmap` の実装を参照) の下に、関手の単純さが失われます。新しい提案された C++ レンジライブラリは、範囲の関手的な性質をはるかに顕著にします。

7.1.7 Reader 関手

あなたがいくらかの直感を得たかもしれない今 — 例えば、関手は何らかのコンテナであるとか、少なくともそれらがパラメータ化されている型の値を含んでいるオブジェクトであるとか — まったく異なるものの例を示しましょう。型 `a` を `a` を返す関数の型への写像を考え

てみてください。私たちは関数型について深くは話していません — その完全な圈論的扱いはこれから来ます — しかし、プログラマとしてそれらについてある程度の理解を持っています。Haskell では、関数型は 2 つの型を取る矢印型コンストラクタ (\rightarrow) を使用して構成されます: 引数型と結果型です。あなたは既にそれを中置形式で見てきました、 $a \rightarrow b$ ですが、括弧を使用すると接頭辞形式でも同様に使用することができます:

```
(→) a b
```

```
Function1[A, B]  
// or  
A => B
```

通常の関数と同様に、2 つ以上の引数を持つ型関数は部分適用することができます。したがって、矢印にただ一つの型引数を与えると、それはまだ別の型 b を必要としています。完全な型 $a \rightarrow b$ を生成します。それが立っている状態では、それは型コンストラクタの全体の族を定義します。 a によってパラメータ化された族です。これが関手の族であるかどうかを見てみましょう。2 つの型パラメータを扱うことは少し混乱するかもしれませんので、いくつかのリネーミングをしましょう。引数型を r と呼び、結果型を a と呼びましょう、これは私たちの以前の関手定義と一致します。それで私たちの型コンストラクタは任意の型 a を型 $r \rightarrow a$ に写像します。それが関手であることを示

すためには、関数 $a \rightarrow b$ を持ち上げて、 $r \rightarrow a$ を取り $r \rightarrow b$ を返す関数を定義したいと思います。これらは、それぞれ、型コンストラクタ (\rightarrow) r が a および b に作用するときに形成される型です。この場合に適用される `fmap` の型シグネチャはこちらです:

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)  
  
def fmap[A, B](f: A => B)(g: R => A): (R => B)
```

次のパズルを解かなければなりません: 関数 $f :: a \rightarrow b$ と関数 $g :: r \rightarrow a$ が与えられたとき、関数 $r \rightarrow b$ を作成します。2つの関数を合成する唯一の方法があり、その結果はちょうど私たちが必要とするものです。それでこちらが私たちの `fmap` の実装です:

```
instance Functor ((->) r) where  
  fmap f g = f . g  
  
  // with the Kind Projector plugin:  
  implicit def function1Functor[R] = new Functor[R => ?] {  
    def fmap[A, B](f: A => B)(g: R => A): (R => B) =  
      f compose g  
  }
```

それはちょうど動作します！あなたが簡潔な表記が好きなら、この定義は合成を接頭辞形式で書き換えることによってさらに短縮することができます:

```
fmap f g = (.) f g
```

```
def fmap[A, B]: (A => B) => (R => A) => (R => B) =  
  f => g => f compose g
```

そして、引数を省略することによって、2つの関数の直接的な等価性をもたらします：

```
fmap = (.)
```

```
def fmap[A, B]: (A => B) => (R => A) => (R => B) =  
  - compose
```

この型コンストラクタ (\rightarrow) r と上記の `fmap` の実装の組み合わせは、reader 関手と呼ばれます。

7.2 関手としてのコンテナ

私たちは、一般的な目的のコンテナ、または少なくともそれらがパラメータ化されている型の値を含む何らかのオブジェクトを定義するプログラミング言語における関手の例をいくつか見てきました。Reader 関手は外れ値のように思えるかもしれません。なぜなら私たちは関数をデータとして考えることはないからです。しかし、私たちは純粋関数がメモ化され得ること、そして関数実行がテーブルルックアップに変換され得ることを見てきました。テーブルはデータです。

逆に、Haskell の遅延評価のために、伝統的なコンテナ、例えばリストは、実際には関数として実装され得ます。例えば、自然数の無限リストは、コンパクトに次のように定義され得ます:

```
nats :: [Integer]
nats = [1..]

// LazyLists are supported as of Scala 2.13
def nats: LazyList[Int] = LazyList.from(1)
```

最初の行では、リストのための Haskell の組み込み型コンストラクタの一つのペアの角括弧があります。2 行目では、リストリテラルを作成するために角括弧が使用されています。明らかに、このような無限リストはメモリに保存され得ません。コンパイラは需要に応じて **Integer** を生成する関数として実装します。Haskell は実効的にデータとコードの区別を曖昧にします。リストは関数と考えられ得、関数は引数を結果にマッピングするテーブルとして考えられ得ます。後者は関数の始域が有限で、かつ大きすぎない場合に実用的でさえあります。しかしながら、**strlen** をテーブルルックアップとして実装することは実用的ではないでしょう。なぜなら、異なる文字列は無限に存在するからです。プログラマとしては、私たちは無限を好みませんが、圏論では無限を朝食にします。それが全ての文字列の集合であろうと、過去、現在、未来の宇宙の全ての可能な状態の集合であろうとー私たちはそれを扱うことができます！ 私は関手オブジェクト(自己関手に

よって生成される型のオブジェクト)を、それがパラメータ化されている型の値または値を含んでいるものと考えるのが好きです。たとえそれらの値が物理的に存在していないとしてもです。関手は C++ の `std::future` のようなものであるかもしれません。それはいつか値を含むかもしれません、それが含まれるとは保証されておらず、そしてあなたがアクセスしようとすると、別のスレッドが実行を終えるのを待つかかもしれません。別の例は Haskell の IO オブジェクトであり、それはユーザー入力を含むか、または「Hello World!」がモニターに表示される未来の宇宙のバージョンを含むかもしれません。この解釈に従うと、関手オブジェクトは、それがパラメータ化されている型の値を含むかもしれないもの、またはそれらの値を生成するためのレシピを含むかもしれないものです。私たちは値にアクセスできるかどうかにはまったく関心がありません。それは完全にオプションであり、関手の範囲外です。私たちが興味を持つのは、関数を使用してこれらの値を操作できること、値にアクセスできる場合は、この操作の結果を見る能够性です。もしそうでなければ、操作が正しく合成され、恒等関数での操作が何も変わらないことだけを気にします。関手オブジェクト内の値にアクセスできるかどうかを気にしないほど、こちらが `a` の引数を完全に無視する型コンストラクタです:

```
data Const c a = Const c
```

```
case class Const[C, A](v: C)
```

`Const` 型コンストラクタは 2 つの型、`c` と `a` を取ります。矢印コンストラクタと同じように、私たちは部分適用することによって関手を作成します。データコンストラクタ (`Const` も呼ばれます) は型 `c` の値をただ一つ取ります。それは `a` に依存しません。この型コンストラクタの `fmap` の型はこちらです:

```
fmap :: (a -> b) -> Const c a -> Const c b
```

```
def fmap[A, B](f: A => B)(ca: Const[C, A]): Const[C, B]
```

関手がその型引数を無視するため、`fmap` の実装はその関数引数を無視する自由があります — 関数が作用するものは何もありません:

```
instance Functor (Const c) where
    fmap _ (Const v) = Const v

implicit def constFunctor[C] = new Functor[Const[C, ?]] {
    def fmap[A, B](f: A => B)(ca: Const[C, A]): Const[C, B] =
        Const(ca.v)
}
```

これは

C++ で少し明確になるかもしれません（私はそのようなことを言う日が来るとは思いませんでした！）、そこでは型引数と値の間には、コンパイル時と実行時というより強い区別があります：

```
template<class C, class A>
struct Const {
    Const(C v) : _v(v) {}
    C _v;
};
```

`fmap` の C++ 実装も関数引数を無視し、本質的に `Const` 引数を再キャストしますが、その値を変更することはありません：

```
template<class C, class A, class B>
Const<C, B> fmap(std::function<B(A)> f, Const<C, A> c) {
    return Const<C, B>{c._v};
}
```

その奇妙さにもかかわらず、`Const` 関手は多くの構成で重要な役割を果たします。圏論では、それは私が以前述べた Δ_c 関手の特別なケースです — ブラックホールの自己関手ケースです。私たちはそれを将来もっと見ることになるでしょう。

7.3 関手の合成

関手が圏間で合成できると自分自身を納得させるのは難しいことはありません。集合間の関数が合成できるのと同じように、関手間でも合成できます。対象に作用するときの合成は、それぞれの対象写像の合成であり、射に作用するときも同様です。2つの関手を通じてジャンプした後、恒等射は恒等射として終わり、射の合成は射の合成として終わります。本当にそれ以上のものはありません。特に、自己関手の合成は容易です。私は `maybeTail` 関数を使って書き直します。

Haskell のリストの組み込み実装を使用します：

```
maybeTail :: [a] -> Maybe [a]
maybeTail [] = Nothing
maybeTail (x:xs) = Just xs

def maybeTail[A]: List[A] => Option[List[A]] = {
    case Nil => None
    case Cons(x, xs) => Some(xs)
}
```

(空のリストコンストラクタは、私たちが `Nil` と呼んでいたものが、空のペアの角括弧 `[]` に置き換えられています。`Cons` コンストラクタは、コロン`:`という中置演算子に置き換えられています。) `maybeTail` の結果は、`a` に作用する 2 つの関手、`Maybe` と `[]` の合成の型です。それぞれの関手は自分のバージョンの `fmap` を装備していますが、もし私た

ちが合成されたもの、つまり **Maybe** リストの内容に関数 **f** を適用したい場合はどうでしょうか？ 私たちは 2 層の関手を突破する必要があります。**fmap** を使って外側の **Maybe** を突破することができます。しかし、私たちは **f** を **Maybe** の内部に送ることはできません。なぜなら **f** はリスト上では機能しないからです。私たちは内部リストに作用するために **(fmap f)** を送らなければなりません。例えば、**Maybe** リストの整数の要素を二乗する方法を見てみましょう：

```
square x = x * x

mis :: Maybe [Int]
mis = Just [1, 2, 3]

mis2 = fmap (fmap square) mis

def square: Int => Int = x => x * x

val mis: Option[List[Int]] =
  Some(Cons(1, Cons(2, Cons(3, Nil)))))

val mis2 = optionFunctor.
  fmap(listFunctor.fmap(square))(mis)
```

コンパイラは、型を分析した後、外側の **fmap** には **Maybe** インスタンスからの実装を使用し、内側のものにはリスト関手の実装を使用する

べきだと判断します。上記のコードが次のように書き換えられること
がすぐには明らかではないかもしれません:

```
mis2 = (fmap . fmap) square mis

def fmap0[A, B]: (A => B) => Option[A] => Option[B] =
  optionFunctor.fmap
def fmapL[A, B]: (A => B) => List[A] => List[B] =
  listFunctor fmap
def fmapC[A, B]: (A => B) => Option[List[A]] => Option[List[B]] =
  fmap0.compose(fmapL)

val mis2 = fmapC(square)(mis)
```

しかし、**fmap** は 1 つの引数の関数として考えることができることを覚
えておいてください:

```
fmap :: (a -> b) -> (f a -> f b)

def fmap[F[_], A, B]: (A => B) => (F[A] => F[B])
```

私たちの場合、2 番目の **fmap** で、(**fmap** . **fmap**) の中の **fmap** は次の
引数を取ります:

```
square :: Int -> Int
```

```
def square: Int => Int
```

そして、次の型の関数を返します:

```
[Int] -> [Int]
```

```
List[Int] => List[Int]
```

最初の `fmap` はその関数を取り、次の型の関数を返します:

```
Maybe [Int] -> Maybe [Int]
```

```
Option[List[Int]] => Option[List[Int]]
```

最後に、その関数は `mis` に適用されます。従って、2つの関手の合成は関手であり、その `fmap` は対応する `fmap` の合成です。圏論に戻って: 関手の合成が結合的であることはかなり明白です(対象の写像は結合的であり、射の写像も結合的です)。そして、すべての圏には自己関手として機能する自明な恒等関手があります: それはすべての対象をそれ自体にマッピングし、すべての射をそれ自体にマッピングします。だから関手は圏のいくつかの圏における射と同じ性質を持っています。しかし、その圏はどのようなものでしょうか? それは圏が対象であり、関手が射である圏でなければなりません。それは圏の圏です。しかし、**全ての** 圏の圏はそれ自体を含む必要があり、我々は「全ての集合の集合」が不可能であることを示す同じ種類の逆説に陥ります。

しかし、対象が集合を形成する(つまり集合よりも大きい何かではない)小さい圏のすべての圏である **Cat** と呼ばれる圏はあります(それは大きいので、自分自身のメンバーにはなれません)。小さい圏とは、無限不可算集合でさえ「小さい」と見なされる圏論において、対象が集合を形成するものです。これらのことにつれて触るのは、私たちが抽象化の多くのレベルで同じ構造が繰り返されていることを認識できるのが非常に素晴らしいからです。私たちは後で見るでしょうが、関手もまた圏を形成します。

7.4 チャレンジ

1. **Maybe** 型コンストラクタを関手に変換することができますか？

両方の引数を無視する次のように定義することによって:

```
fmap _ _ = Nothing
```

(ヒント: 関手則をチェックしてください。)

2. **reader** 関手の関手則を証明してください。ヒント: それは本当にシンプルです。
3. あなたの 2 番目に好きな言語(もちろん 1 番目は Haskell です)で **reader** 関手を実装してください。
4. リスト関手の関手則を証明してください。リストのテール部分にそれを適用するときに規則が真であると仮定します(言い換えれば、**帰納法**を使ってください)。

8

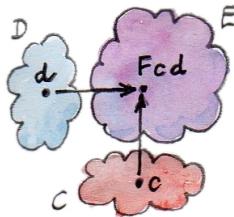
関手性

関手が何であるか理解した今、小さな関手から大きな関手を構成する方法を見てみましょう。特に、圏間の対象に対応する型コンストラクタが、射に対応するマッピングも含む関手に拡張できるかどうかを見るのが興味深いです。

8.1 双関手

関手は Cat (圏の圏) の射なので、射について、特に関数についての直感が関手にも適用されます。例えば、2つの引数を取る関数があるように、2つの引数を取る関手、すなわち**双関手**が存在します。双関

手は、圏 C の対象と圏 D の対象の各ペアを、圏 E の対象に写像します。これは、圏 $C \times D$ から E への写像と言っているのと同じです。



これは比較的わかりやすいですが、関手性により、双関手は射も写像する必要があります。ただし、今回は C からの射と D からの射のペアを E の射に写像する必要があります。

再び、射のペアは単なる圏 $C \times D$ から E への单一の射です。圏のデカルト積における射を定義すると、一つの対象ペアから別の対象ペアへ行く射のペアです。これらの射ペアは明らかな方法で合成されます：

$$(f, g) \circ (f', g') = (f \circ f', g \circ g')$$

合成は結合的であり、恒等射ペア (id, id) が存在します。したがって、圏のデカルト積もまた圏です。

双関手についてもう少し簡単に考える方法は、それをそれぞれの引数に対して関手として考えることです。したがって、関手から双関手への関手則 – 結合性と恒等性を持つこと – を翻訳する代わりに、それぞれの引数に対して別々にそれらをチェックするだけで十分です。しかし、一般的には、個別の関手性だけでは共同関手性を証明するに

は不十分です。共同関手性が失敗する圏はプレモノイダルと呼ばれます。

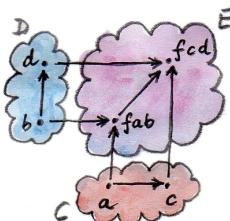
Haskell で双関手を定義しましょう。この場合、すべての三つの圏は同じです: Haskell 型の圏。双関手は 2 つの型引数を取る型コンストラクタです。ここに `Control.Bifunctor` ライブラリから直接取った `Bifunctor` 型クラスの定義があります:

```
class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
    bimap g h = first g . second h
    first :: (a -> c) -> f a b -> f c b
    first g = bimap g id
    second :: (b -> d) -> f a b -> f a d
    second = bimap id

trait Bifunctor[F[_], _] {
    def bimap[A, B, C, D](g: A => C)(h: B => D): F[A, B] => F[C, D] =
        first(g) compose second(h)

    def first[A, B, C](g: A => C): F[A, B] => F[C, B] =
        bimap(g)(identity[B])

    def second[A, B, D](h: B => D): F[A, B] => F[A, D] =
        bimap(identity[A])(h)
}
```



bimap

型変数 **f** が双関手を表しています。すべての型シグネチャにおいて、常に 2 つの型引数に適用されていることがわかります。最初の型シグネチャは **bimap** を定義しています: 一度に 2 つの関数のマッピングです。結果は持ち上げられた関数、(**f a b -> f c d**) で、双関手の型コンストラクタによって生成された型に作用します。**bimap** の **Control.Bifunctor** と **second** に関するデフォルト実装があります。(前に述べたように、これは常に機能するとは限らず、二つのマップが可換でない場合があります、つまり **first g . second h** が **second h . first g** と同じでないかもしれません。)

他の 2 つの型シグネチャ、**first** と **second** は、それぞれ最初と二番目の引数における **f** の関手性を証明する **fmap** の二つです。



型クラス定義は、`bimap` についてのデフォルト実装の両方を提供します。

`Bifunctor` のインスタンスを宣言するとき、`bimap` を実装して `first` と `second` のデフォルトを受け入れるか、`first` と `second` の両方を実装して `bimap` のデフォルトを受け入れるかの選択があります（もちろん、すべての三つを実装することもできますが、その場合はそれらがこのように関連していることを自分で確認する必要があります）。

8.2 積と余積の双関手

双関手の重要な例は、普遍構成で定義される圏の積一二つの対象の積です。任意の対象ペアに対して積が存在する場合、それらの対象から積へのマッピングは双関手的です。これは一般的にも、特に Haskellにおいても真です。ここにペアコンストラクター最も単純な積型のための `Bifunctor` インスタンスがあります：

```
instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)

implicit val tuple2Bifunctor = new Bifunctor[Tuple2] {
    override def bimap[A, B, C, D](f: A => C)(g: B => D): ((A,
        => B)) => (C, D) = {
        case (x, y) => (f(x), g(y))
    }
}
```

選択肢はほとんどありません: `bimap` は単に最初の関数をペアの最初のコンポーネントに適用し、二番目の関数を二番目のコンポーネントに適用します。コードは、型が与えられると自動的に書かれます:

```
bimap :: (a -> c) -> (b -> d) -> (a, b) -> (c, d)

def bimap[A, B, C, D](f: A => C)(g: B => D): ((A, B)) => (C, D)
```

この場合、双関手の作用は型のペアを作ることです。例えば:

```
(,) a b = (a, b)
```

双対性によって、余積も、圏内の任意の対象ペアに対して定義されていれば、双関手です。Haskell では、`Either` 型コンストラクタが `Bifunctor` のインスタンスであることによって示されています:

```
instance Bifunctor Either where
    bimap f _ (Left x) = Left (f x)
    bimap _ g (Right y) = Right (g y)

implicit val eitherBifunctor = new Bifunctor[Either] {
    override def bimap[A, B, C, D](f: A => C)(g: B => D):
        Either[A, B] => Either[C, D] = {
        case Left(x) => Left(f(x))
        case Right(y) => Right(g(y))
    }
}
```

このコードも自動的に書かれます。

さて、モノイダル圏について話したこと覚えていませんか？ モノイダル圏は、対象に作用する二項演算子と、単位対象を定義します。Setがデカルト積に関してモノイダル圏であり、単集合を単位としていること、また、非交和に関しても空集合を単位とするモノイダル圏であることを言及しました。言及していないのは、モノイダル圏にとって二項演算子が双関手であるという要求です。これは非常に重要な要求です - 私たちはモノイダル積が、射によって定義される圏の構造と互換性を持っていることを望みます。これで、モノイダル圏の完全な定義に一步近づきました（自然性について学ぶ前にはそこに到達できませんが）。

8.3 関手的な代数的データ型

私たちは、**fmap** を定義できるパラメータ化されたデータ型のいくつかの例を見てきました — それらは関手でした。複雑なデータ型は、単純なデータ型から構成されます。特に、代数的データ型 (ADT) は、和と積を使用して作成されます。私たちはすでに和と積が関手的であることを見てきました。また、関手は合成可能です。したがって、ADT の基本的な構成要素が関手的であることを示すことができれば、パラメータ化された ADT も関手的であるとわかります。

では、パラメータ化された代数的データ型の構成要素は何でしょうか？ まず、関手の型パラメーターに依存しない項目があります。例えば、**Maybe** の **Nothing** や **List** の **Nil** です。これらは **Const** 関手に相当します。**Const** 関手はその型パラメーターを無視します（実際には、私たちにとって興味のある二番目の型パラメーターを無視します。最初のものは一定です）。

次に、型パラメーター自身を単純にカプセル化する要素があります。例えば、**Maybe** の **Just** です。これらは恒等関手に相当します。以前、恒等関手を *Cat* の恒等射として言及しましたが、Haskell での定義は示しませんでした。それはこちらです：

```
data Identity a = Identity a
```

```
type Id[A] = A
```

```
instance Functor Identity where
    fmap f (Identity x) = Identity (f x)

implicit val identityFunctor = new Functor[Id] {
    def fmap[A, B](f: A => B)(x: Id[A]): Id[B] =
        f(x)
}
```

`Identity` を、型 `a` の常に 1 つの(不变な) 値を格納する最も単純なコレクションと考えることができます。

その他のすべての代数的データ構造は、積と和を使って、この 2 つのプリミティブな要素から構成されます。

この新しい知識を持って、`Maybe` 型コンストラクタを新たな視点から見てみましょう:

```
data Maybe a = Nothing | Just a

sealed trait Option[+A]
case object None extends Option[Nothing]
case class Some[A](a: A) extends Option[A]
```

これは 2 つの型の和ですが、今では和が関手的であることがわかります。最初の部分、`Nothing` は、`a` に作用する `Const ()` として表現できます (`Const` の最初の型パラメータは単位に設定されています - 後で `Const` のより興味深い使用法を見ていきます)。二番目の部分は、単に

恒等関手の異なる名前です。私たちは、`Maybe` を同型まで `Either` と 2 つの関手、`Const ()` と `Identity` の合成として定義できました:

```
type Maybe a = Either (Const () a) (Identity a)
```

```
type Option[A] = Either[Const[Unit, A], Id[A]]
```

したがって、`Maybe` は、2 つの関手、`Const ()` と `Identity` で部分適用された双関手 `Either` の合成です。`(Const` は本当に双関手ですが、ここでは常に部分適用されています。)

私たちはすでに、関手の合成が関手であることを見てきました - 双関手と 2 つの関手の合成が射に対してどのように機能するかを理解するだけです。2 つの射を与えられたら、1 つを 1 つの関手で、もう 1 つを別の関手で持ち上げます。そして、双関手で持ち上げられた射のペアを持ち上げます。

Haskell でこの合成を表現しましょう。新しいデータ型を定義しますが、それは双関手 `bf` (2 つの型引数を取る型コンストラクタである型変数)、2 つの関手 `fu` と `gu` (それぞれ 1 つの型変数を取る型コンストラクタ)、そして 2 つの通常の型 `a` と `b` によってパラメータ化されています。私たちは `fu` を `a` に適用し、`gu` を `b` に適用し、その後で `bf` を結果の 2 つの型に適用します:

```
newtype BiComp bf fu gu a b = BiComp (bf (fu a) (gu b))
```

```
case class BiComp[BF[_], _], FU[_], GU[_], A, B](v: BF[FU[A], GU[B]])
```

これが対象、つまり型上の合成です。Haskell では型コンストラクタを型に適用する方法は、関数を引数に適用する方法と同じです。

もし少し迷っているなら、`BiComp` を `Either`、`Const ()`、`Identity`、`a`、そして `b` に、この順番で適用してみてください。そうすると、私たちの `Maybe b` の骨組みバージョンを回復することができます (`a` は無視されます)。

新しいデータ型 `BiComp` は、`a` と `b` において双関手ですが、`bf` 自体が `Bifunctor` であり、`fu` と `gu` が `Functor` である場合に限ります。コンパイラは `bf` に対して `bimap` の定義が、そして `fu` と `gu` に対して `fmap` の定義が存在することを知っていなければなりません。Haskell では、これはインスタンス宣言における前提条件として表現されます: 一連のクラス制約に続いて二重射があります:

```
instance (Bifunctor bf, Functor fu, Functor gu) =>
  Bifunctor (BiComp bf fu gu) where
    bimap f1 f2 (BiComp x) = BiComp (bimap (fmap f1) (fmap
      ↵ f2) x)

implicit def bicompBifunctor[
  BF[_], _], FU[_], GU[_]](
  implicit BF: Bifunctor[BF],
```

```
FU: Functor[FU], GU: Functor[GU]) = {
    // partially-applied type BiComp
    type BiCompAB[A, B] = BiComp[BF, FU, GU, A, B]
    new Bifunctor[BiCompAB] {
        override def bimap[A, B, C, D](f1: A => C)(f2: B => D):
            BiCompAB[A, B] => BiCompAB[C, D] = {
            case BiComp(x) =>
                BiComp(
                    BF.bimap(FU fmap(f1))(GU fmap(f2))(x)
                )
            }
        }
    }
}
```

BiComp の bimap の実装は、bf の bimap と 2 つの fmap の用語で与えられます。コンパイラは自動的にすべての型を推測し、BiComp が使用されるたびに正しいオーバーロードされた関数を選択します。

bimap の定義における x は次の型を持ちます:

bf (fu a) (gu b)

BF[FU[A], GU[B]]

これはかなり多くのものです。外側の bimap は外側の bf 層を通過し、2 つの fmap はそれぞれ fu と gu の下を掘ります。もし f1 と f2 の型が:

```
f1 :: a -> a'
```

```
f2 :: b -> b'
```

```
def f1: A => A1
```

```
def f2: B => B1
```

ならば、最終結果は型 `bf (fu a') (gu b')` になります:

```
bimap :: (fu a -> fu a') -> (gu b -> gu b')
```

```
-> bf (fu a) (gu b) -> bf (fu a') (gu b')
```

```
def bimap: (FU[A] => FU[A1]) => (GU[B] => GU[B1]) =>
```

```
↪ BiComp[BF, FU, GU, A, B] => BiComp[BF, FU, GU, A1, B1]
```

もしジグソーパズルが好きなら、このような型操作で何時間も楽しむことができるでしょう。

したがって、`Maybe` が関手であることを証明する必要はなかったことがわかります - これは、2つの関手的原始からの和として構成された方法から導かれました。

洞察に富んだ読者は次のような質問をするかもしれません: 代数的データ型のための `Functor` インスタンスの導出がこれほど機械的なら、コンパイラによって自動化され、実行されることはできないのでしょうか? 実際にはでき、されています。あなたはソースファイルの

先頭にこの行を含めることによって特定の Haskell 拡張機能を有効にする必要があります:

```
{-# LANGUAGE DeriveFunctor #-}
```

そして、データ構造に **deriving Functor** を追加します:

```
data Maybe a = Nothing | Just a deriving Functor
```

そして対応する **fmap** があなたのために実装されます。

代数的データ構造の規則性は、**Functor** だけでなく、以前言及した **Eq** 型クラスを含むいくつかの他の型クラスのインスタンスも導出することが可能になります。また、独自の型クラスのインスタンスをコンパイラに導出させる方法を教えるオプションもありますが、それは少し高度です。しかし、考え方は同じです: 基本的な構成要素と和と積のための動作を提供し、残りはコンパイラに任せます。

8.4 C++ における関手

もし C++ プログラマであれば、関手を実装することに関しては自分自身で行うことになるでしょう。しかし、C++ において代数的データ構造のいくつかの型を認識することができるはずです。そのようなデータ構造がジェネリックテンプレートにされた場合、それに対して **fmap** を迅速に実装することができるはずです。

木データ構造を見てみましょう。これは Haskell では再帰的な和型として定義されるでしょう:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
    deriving Functor

sealed trait Tree[+A]
case class Leaf[A](a: A) extends Tree[A]
case class Node[A](l: Tree[A], r: Tree[A]) extends Tree[A]
// implicit def treeFunctor = ???
```

前に言及したように、和型を C++ で実装する一つの方法はクラス階層を通じてです。オブジェクト指向言語で自然なのは、基底クラス **Functor** の仮想関数として **fmap** を実装し、すべてのサブクラスでそれをオーバーライドすることです。残念ながら、これは **fmap** がテンプレートであり、それが作用するオブジェクトの型 (**this** ポインター) だけでなく、それに適用された関数の戻り型によってもパラメータ化されているため、不可能です。C++ では、仮想関数はテンプレート化できません。私たちは **fmap** をジェネリックな自由関数として実装し、**dynamic_cast** でパターンマッチングを置き換えます。

基底クラスは少なくとも 1 つの仮想関数を定義する必要があります。動的キャストをサポートするために、私たちはデストラクターを仮想にします (いずれにせよ、良いアイデアです):

```
template<class T>
struct Tree {
    virtual ~Tree() {}
};
```

`Leaf` は単なる `Identity` 関手の変形です:

```
template<class T>
struct Leaf : public Tree<T> {
    T _label;
    Leaf(T l) : _label(l) {}
};
```

`Node` は積型です:

```
template<class T>
struct Node : public Tree<T> {
    Tree<T> * _left;
    Tree<T> * _right;
    Node(Tree<T> * l, Tree<T> * r) : _left(l), _right(r) {}
};
```

`fmap` を実装するときには、`Tree` の型に対する動的ディスパッチを利用します。`Leaf` ケースは `Identity` 版の `fmap` を適用し、`Node` ケースは 2 つのコピーの `Tree` 関手で合成された双関手のように扱われます。C++ プログラマとして、このような用語でコードを分析することに慣れていないかもしれませんのが、圏論的な思考における良い練習です。

```
template<class A, class B>
Tree<B> * fmap(std::function<B(A)> f, Tree<A> * t) {
    Leaf<A> * pl = dynamic_cast <Leaf<A>*>(t);
    if (pl)
        return new Leaf<B>(f (pl->_label));
    Node<A> * pn = dynamic_cast<Node<A>*>(t);
    if (pn)
        return new Node<B>( fmap<A>(f, pn->_left)
                            , fmap<A>(f, pn->_right));
    return nullptr;
}
```

簡単のために、私はメモリとリソース管理の問題を無視することにしましたが、実際のコードではおそらくスマートポインターを使用するでしょう（共有か非共有かはあなたの方針次第）。

Haskell の `fmap` の実装と比較してみてください：

```
instance Functor Tree where
    fmap f (Leaf a) = Leaf (f a)
    fmap f (Node t t') = Node (fmap f t) (fmap f t')

implicit val treeFunctor = new Functor[Tree] {
    def fmap[A, B](f: A => B)(fa: Tree[A]): Tree[B] = fa match {
        case Leaf(a) => Leaf(f(a))
        case Node(t, t1) => Node(fmap(f)(t), fmap(f)(t1))
    }
}
```

この実装もまた、コンパイラによって自動的に導出されることがあります。

8.5 Writer 関手

以前に紹介した Kleisli 圏に戻ることを約束しました。その圏の射は、Writer データ構造を返す「装飾された」関数として表されました。

```
type Writer a = (a, String)
```

```
type Writer[A] = (A, String)
```

その装飾が何らかの形で自己関手に関連していると言いました。そして、実際に Writer 型コンストラクタは a において関手的です。それに対して fmap を実装する必要すらありませんでした。なぜなら、それは単なる単純な積型だからです。

しかし、一般的には、Kleisli 圏と関手の関係は何でしょうか？ Kleisli 圏は圏であり、合成と恒等性を定義します。合成は魚演算子によって与えられます：

```
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
m1 >=> m2 = \x ->
  let (y, s1) = m1 x
    (z, s2) = m2 y
  in (z, s1 ++ s2)
```

```
object kleisli {  
    //allows us to use >=> as an infix operator  
    implicit class KleisliOps[A, B](m1: A => Writer[B]) {  
        def >=>[C](m2: B => Writer[C]): A => Writer[C] =  
            x => {  
                val (y, s1) = m1(x)  
                val (z, s2) = m2(y)  
                (z, s1 + s2)  
            }  
    }  
}
```

そして、`return` と呼ばれる関数による恒等射があります:

```
return :: a -> Writer a  
return x = (x, "")  
  
// return is a keyword in Scala  
def pure[A](x: A): Writer[A] = (x, "")
```

実は、これら 2 つの関数の型を十分に長い間 (本当に、長い間) 見ていれば、`fmap` として機能する正しい型シグネチャの関数を生成する方法を見つけることができます。このように:

```
fmap f = id >=> (\x -> return (f x))
```

```
import kleisli._

def fmap[A, B](f: A => B): Writer[A] => Writer[B] =
    identity[Writer[A]] _ >=> (x => pure(f(x)))
```

ここでは、魚演算子は 2 つの関数を組み合わせています: 1 つはおなじみの `id`、もう 1 つは `return` を `f` のラムダの引数に適用するラムダです。`id` の使用について頭を悩ませるかもしれません。魚演算子の引数は「通常の」型から装飾された型に変換する関数ではないのでしょうか？ 実際にはそうではありません。`a` の `a -> Writer b` は「通常の」型である必要はありません。それは型変数なので、何でもよく、特に `Writer b` のような装飾された型になることができます。

したがって、`id` は `Writer a` を受け取り、`Writer a` に変換します。魚演算子は `a` の値を釣り出し、ラムダに `x` として渡します。そこで `f` はそれを `b` に変換し、`return` はそれを装飾し、`Writer b` にします。すべてを一緒に考えると、`Writer a` を取り、`Writer b` を返す関数が得られます。これはちょうど `fmap` が生成すべきものです。

この議論は非常に一般的です: あなたは `Writer` を任意の型コンストラクタに置き換えることができます。魚演算子と `return` をサポートする限り、`fmap` を定義することができます。したがって、Kleisli 圈の装飾は常に関手です。(しかし、すべての関手が Kleisli 圈を生み出すわけではありません。)

あなたは、私たちが定義した `fmap` が、`deriving Functor` でコンパイラが私たちのために導出してくれる `fmap` と同じものかどうか疑問

に思うかもしれません。興味深いことに、それはそうです。これは、Haskell が多相的関数を実装する方法によるものです。それはパラメトリック多相性と呼ばれ、いわゆる無料の定理の源です。そのような定理の 1 つは、与えられた型構造に対して恒等性を保つ `fmap` の実装がある場合、それは一意でなければならないというものです。

8.6 共変関手と反変関手

`writer` 関手を見直したので、`reader` 関手に戻ってみましょう。それは部分適用された関数矢印型構造に基づいていました:

```
(->) r  
  
// with Kind Projector plugin  
Function1[R, ?] or R => ?
```

これを型シノニムとして書き換えることができます:

```
type Reader r a = r -> a  
  
type Reader[R, A] = R => A
```

私たちが以前見たように、そのための `Functor` インスタンスは次のようにになります:

```
instance Functor (Reader r) where
    fmap f g = f . g

    implicit def readerFunctor[R] = new Functor[Reader[R, ?]] {
        def fmap[A, B](f: A => B)(g: Reader[R, A]): Reader[R, B] =
            f compose g
    }
```

しかし、ペア型コンストラクタや `Either` 型コンストラクタのように、関数型コンストラクタは 2 つの型引数を取ります。ペアと `Either` は両方の引数において関手的でした - それらは双関手でした。関数コンストラクタも双関手ですか？

最初の引数において関手的にすることを試みましょう。型シノニムを使いましょう。これは `Reader` と似ていますが、引数が反転しています:

```
type Op[r] a = a -> r

type Op[R, A] = A => R
```

この場合、戻り値の型 `r` を固定し、引数の型 `a` を変えます。`fmap` を実装しようとすると、それは以下のような型シグネチャを持つことになります:

```
fmap :: (a -> b) -> (a -> r) -> (b -> r)
```

```
def fmap[A, B]: (A => B) => (A => R) => (B => R)
```

aを取り、それぞれ **b** と **r** を返す 2 つの関数を持っているだけでは、**b**を取り **r** を返す関数を構成する方法は単純にはありません！しかし、もし最初の関数を何とか反転させることができれば、つまりそれが **b**を取り **a** を返すようにできれば、異なるものが得られるかもしれません。任意の関数を反転することはできませんが、反対の圏に行くことはできます。

簡単に復習しましょう：圏 **C** ごとに、その双対圏 **C^{op}** があります。これは **C** と同じ対象を持ちますが、すべての射が逆転しています。

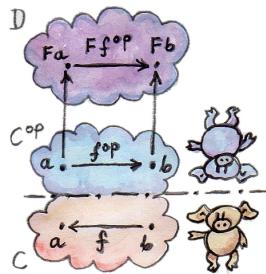
C^{op} と他の圏 **D** との間の関手を考えてみましょう：

$$F :: \mathbf{C}^{op} \rightarrow \mathbf{D}$$

このような関手は、**C^{op}** 内の射 $f^{op} :: a \rightarrow b$ を **D** 内の射 $Ff^{op} :: Fa \rightarrow Fb$ に写像します。しかし、射 f^{op} は秘密裏に元の圏 **C** のある射 $f :: b \rightarrow a$ に対応しています。方向の逆転に注目してください。

ここで、 F は通常の関手ですが、 F に基づいて別の写像を定義することができます。それを G としましょう。それは **C** から **D** への写像です。それは対象を F と同じ方法で写像しますが、射を写像するときはそれらを逆転します。それは **C** の射 $f :: b \rightarrow a$ を取り、それを反対の射 $f^{op} :: a \rightarrow b$ に写像し、その後それに関手 F を適用して、 $Ff^{op} :: Fa \rightarrow Fb$ を得ます。

Fa が Ga と同じであり、 Fb が Gb と同じであると考えると、全体のプロセスは次のように記述できます: $Gf :: (b \rightarrow a) \rightarrow (Ga \rightarrow Gb)$ これは「捩れのある関手」です。このように射の方向を逆転させる圏の写像は、**反変関手**と呼ばれます。反変関手は、反対圏からの通常の関手に過ぎません。ちなみに、これまでに研究してきた通常の関手は、**共変関手**と呼ばれます。



反変関手(実際には、反変自己関手)を定義する Haskell の型クラスがこちらです:

```
class Contravariant f where
    contramap :: (b -> a) -> (f a -> f b)

trait Contravariant[F[_]] {
    def contramap[A, B](f: B => A)(fa: F[A]): F[B]
}
```

私たちの型コンストラクタ `Op` はそのインスタンスです:

```
instance Contravariant (Op r) where
    -- (b -> a) -> Op r a -> Op r b
    contramap f g = g . f

implicit def opContravariant[R] = new Contravariant[R, ?] {
    def contramap[A, B](f: B => A)(g: Op[R, A]): Op[R, B] =
        g compose f
}
```

関数 `f` は、`Op` の内容の前(つまり右)に挿入されることに注意してください。

`Op` の `contramap` の定義は、引数を反転する特別な関数である `flip` を使用することでさらに簡潔にすることができます:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f y x = f x y

def flip[A, B, C]: (A => B => C) => (B => A => C) =
    f => y => x => f(x)(y)
```

それを使って、次のようにになります:

```
contramap = flip (.)

def compose[A, B, C]: (A => B) => (C => A) => C => B =
    f => g => c => f(g(c))
```

```
def contramap[A, B, R](f: B => A)(g: Op[R, A]): Op[R, B] =  
  flip(compose[A, R, B])(f)(g)  
  // or just: (f andThen g)
```

8.7 プロ関手

関数矢印演算子は、その最初の引数において反変的であり、二番目の引数において共変的であることがわかりました。このような性質を持つものに名前はありますか？ 対象圏が `Set` である場合、このようなものはプロ関手と呼ばれます。反変関手が反対圏からの共変関手と同等であるため、プロ関手は次のように定義されます：

$$\mathbf{C}^{op} \times \mathbf{D} \rightarrow \mathbf{Set}$$

第一近似として、Haskell の型は集合であるため、2つの引数を持つ型コンストラクタ `p` に `Profunctor` という名前を適用します。それは最初の引数において反関手的であり、二番目の引数において関手的です。こちらが `Data.Profunctor` ライブラリからの適切な型クラスです：

```
class Profunctor p where  
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d  
  dimap f g = lmap f . rmap g  
  lmap :: (a -> b) -> p b c -> p a c  
  lmap f = dimap f id  
  rmap :: (b -> c) -> p a b -> p a c
```

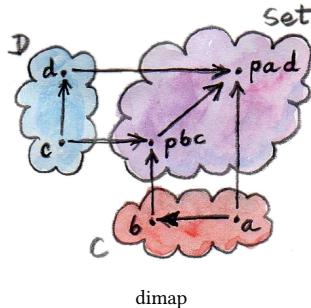
```
rmap = dimap id

trait Profunctor[F[_], _] {
  def dimap[A, B, C, D]: (A => B) => (C => D) => F[B, C] =>
    F[A, D] = f => g =>
    lmap(f) compose rmap[B, C, D](g)

  def lmap[A, B, C]: (A => B) => F[B, C] => F[A, C] = f =>
    dimap(f)(identity[C])

  def rmap[A, B, C]: (B => C) => F[A, B] => F[A, C] =
    dimap[A, A, B, C](identity[A])
}
```

すべての 3 つの関数はデフォルト実装付きです。**Bifunctor** と同じように、**Profunctor** のインスタンスを宣言する際には、**dimap** を実装して **lmap** と **rmap** のデフォルトを受け入れるか、または **lmap** と **rmap** の両方を実装して **dimap** のデフォルトを受け入れる選択があります。



これで、関数矢印演算子が **Profunctor** のインスタンスであると断言することができます:

```

instance Profunctor (->) where
    dimap ab cd bc = cd . bc . ab
    lmap = flip (.)
    rmap = (.)

implicit val function1Profunctor = new Profunctor[Function1]
{
    override def bimap[A, B, C, D]: (A => B) => (C => D) => (B
        => C) => (A => D) = ab => cd => bc =>
        cd compose bc compose ab

    override def lmap[A, B, C]: (A => B) => (B => C) => (A =>
        C) =
        flip(compose)
}

```

```

override def rmap[A, B, C]: (B => C) => (A => B) => (A =>
  ↵  C) =
  compose
}

```

プロ関手は、Haskell の lens ライブラリでの応用があります。私たちは、エンドと余エンドについて話すときに再びそれらを見るでしょう。

8.8 ホム関手

上記の例は、対象のペア a と b を取り、それらの間の射の集合、ホム集合 $\mathbf{C}(a, b)$ に割り当てる写像が関手であるというより一般的な主張の反映です。それは $\mathbf{C}^{op} \times \mathbf{C}$ から集合の圏 \mathbf{Set} への関手です。

射に対するその作用を定義しましょう。 $\mathbf{C}^{op} \times \mathbf{C}$ の射は \mathbf{C} からの射のペアです:

$$\begin{aligned} f &:: a' \rightarrow a \\ g &:: b \rightarrow b' \end{aligned}$$

このペアを持ち上げることは、集合 $\mathbf{C}(a, b)$ から集合 $\mathbf{C}(a', b')$ への射(関数)でなければなりません。 $\mathbf{C}(a, b)$ の任意の要素 h (それは a から b への射です)を取り、それに次のものを割り当てます:

$$g \circ h \circ f$$

これは $\mathbf{C}(a', b')$ の要素です。

ご覧の通り、ホム関手はプロ関手の特別なケースです。

8.9 チャレンジ

1. データ型が双関手であることを示してください:

```
data Pair a b = Pair a b
```

追加のクレジットとして、**Bifunctor** のすべての 3 つの方法を実装し、これらの定義が適用可能な場合にデフォルト実装と互換性があることを等式推論を使用して示してください。

2. **Maybe** の標準定義とこの糖衣構文との間の同型を示してください:

```
type Maybe' a = Either (Const () a) (Identity a)
```

ヒント: 2 つの実装間にマッピングを定義します。追加のクレジットとして、等式推論を使用してそれらが互いに逆であることを示してください。

3. 別のデータ構造を試してみましょう。私はそれを **PreList** と呼びます、なぜならそれは **List** の前身です。それは再帰を型パラメーター **b** で置き換えます。

```
data PreList a b = Nil | Cons a b
```

`List` を再帰的に `PreList` に適用することで、私たちの以前の `List` の定義を回復することができます (固定点について話すときにはどのように行うかを見ます)。

`PreList` が `Bifunctor` のインスタンスであることを示してください。

4. 次のデータ型が `a` と `b` において双関手を定義することを示してください:

```
data K2 c a b = K2 c
```

```
data Fst a b = Fst a
```

```
data Snd a b = Snd b
```

追加のクレジットとして、あなたのソリューションを Conor McBride の論文 [Clowns to the Left of me, Jokers to the Right^{*1}](#) と照合してください。

5. Haskell 以外の言語で双関手を定義してください。その言語で汎用ペアに対して `bimap` を実装してください。
6. `std::map` は 2 つのテンプレート引数 `Key` と `T` において双関手またはプロ関手と考えるべきでしょうか？ このデータ型をどのように再設計するとそうなりますか？

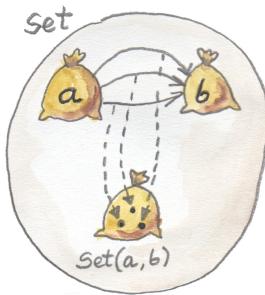
^{*1} <http://strictlypositive.org/CJ.pdf>

9

関数型

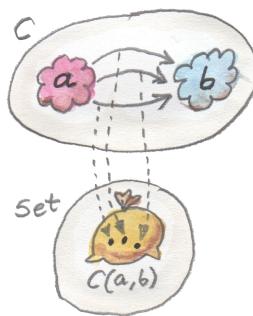
これまで私は関数型の意味を大まかに説明してきました。関数型は他の型とは異なります。

例えば **Integer** を考えてみましょう。これは単に整数の集合です。**Bool** は二要素集合です。しかし、関数型 $a \rightarrow b$ はそれ以上のものです。それは、対象 a と b の間の射の集合です。任意の圏の二つの対象間の射の集合はホム集合と呼ばれます。圏 **Set** では、すべてのホム集合が自身が属する同じ圏の対象であることがよくあります — 結局のところ、それは集合ですから。



Set 内のホム集合はただの集合です

他の圏ではホム集合は圏外部のものです。それらは**外部ホム集合**とさえ呼べれます。



圏 C 内のホム集合は外部集合です

関数型を特別なものにしているのは、**Set** 圏の自己参照的な性質です。しかし、少なくともいくつかの圏では、ホム集合を表す対象を構成する方法があります。そのような対象は内部ホム集合と呼ばれます。

9.1 普遍構成

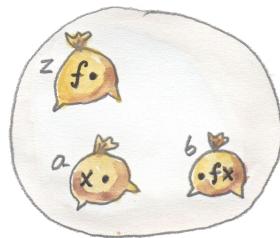
一時的に関数型が集合であるということを忘れて、関数型、または一般的に内部ホム集合を最初から構成してみましょう。通常のように、**Set** 圏から手がかりを得ますが、集合の特性を避けるように注意して、構成が他の圏にも自動的に適用されるようにします。

関数型は、引数型と結果型との関係のために、複合型と見なされるかもしれません。私たちはすでに対象間の関係を含む複合型の構成を見てきました。普遍構成を使って**積型**と**余積型**を定義しました。同じトリックを使って関数型を定義することができます。我々は、構成している関数型、引数型、および結果型の 3 つの対象を含むパターンが必要です。

これらの 3 つの型を接続する明白なパターンは、**関数適用**または**評価**と呼ばれます。関数型の候補としましょう z (圏 **Set** にいない場合、これは他のどの対象とも同様に单なる対象です) と引数型 a (対象)、適用はこのペアを結果型 b (対象) にマッピングします。我々は 3 つの対象、そのうち 2 つは固定されています (引数型と結果型を表すもの)。

また、我々は適用というマッピングを持っています。対象の内部を見ることができれば、 z の関数 f (要素) を a の引数 x (要素) とペアに

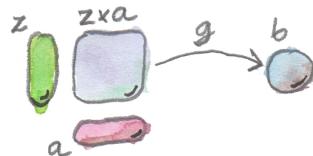
し、それを fx (f による x の適用、 b の要素) にマッピングすることができます。



Set では関数の集合 z から関数 f を選び、集合(型) a から引数 x を選ぶことができます。集合(型) b における要素 fx を得ます。

しかし、個々のペア (f, x) を扱う代わりに、関数型 z と引数型 a の全積について話すこともできます。積 $z \times a$ は対象であり、 b への適用射として、 g という射を選ぶことができます。Set では、 g はすべてのペア (f, x) を fx にマッピングする関数です。

パターンは次のようになります: 2つの対象 z と a の積が、 g という射によって別の対象 b に接続されています。

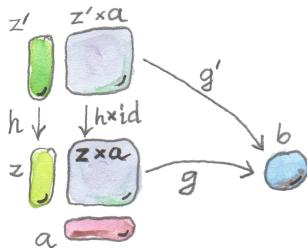


普遍構成の出発点となる対象と射のパターン

このパターンは、普遍構成を使って関数型を特定するのに十分具体的ですか？すべての圏ではありません。しかし私たちが興味を持っている圏ではそうです。別の質問として、まず積を定義せずに関数対象を定義することは可能でしょうか？すべての対象のペアに対して積が存在しない圏もあります。答えはいいえです：積型がなければ、関数型は存在しません。指数について話すとき、これについて後で戻ってきます。

普遍構成を見直しましょう。我々は対象と射のパターンから始めます。それが我々の漠然としたクエリであり、通常はたくさんのヒットを生み出します。特に **Set** では、ほぼすべてが何かに接続されています。任意の対象 z を取り、 a とその積を形成し、 b への関数が存在します (b が空集合でない限り)。

それが私たちが秘密兵器を適用するときです：ランキング。これは通常、候補の対象間に何らかの方法で因数分解する一意のマッピングがあることを要求することによって行われます。この場合、 $z \times a$ から b への射 g を持つ z と、 g を介して g' の適用が因数分解されるような方法で z へ一意にマップされる z' がある場合、 z と g が他の z' との独自の適用 g' よりも優れていると判断します。(ヒント：写真を見ながらこの文を読んでください。)



関数対象の候補間のランキングを確立する

ここでトリッキーな部分です、そしてこの特定の普遍構成を今まで延期した主な理由です。射 $h :: z' \rightarrow z$ を与えられた場合、 a で交差する z' と z の両方を含む図を閉じたいと思います。 z' から z へのマッピング h が与えられたとき、私たちは $z' \times a$ から $z \times a$ へのマッピングが必要です。そして今、**積の関手性**を議論した後、それを行う方法を知っています。積自体が関手（より正確にはエンド・バイ・関手）であるため、射のペアを持ち上げることが可能です。つまり、対象だけでなく射の積も定義できます。

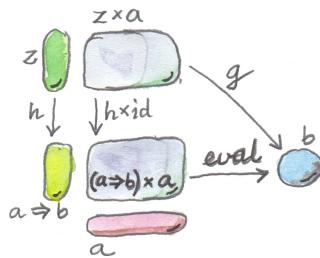
積の第二コンポーネント $z' \times a$ を触れずに、射のペア (h, \mathbf{id}) を持ち上げるつもりです。ここで **id** は a 上の恒等写像です。

したがって、 g の適用を別の適用 g' からどのように因数分解できるかが分かります。

$$g' = g \circ (h \times \mathbf{id})$$

ここでの鍵は射上の積の作用です。

普遍構成の第三部分は、普遍的に最も優れた対象を選択することです。この対象を $a \Rightarrow b$ と呼びましょう（これを一つの対象に対する象徴的な名前として考えてください、Haskell の型クラス制約と混同しないでください—後でそれを名付けるさまざまな方法について話し合います）。この対象は、 $(a \Rightarrow b) \times a$ から b への適用 —*eval* と呼ばれる射—を持っています。対象 $a \Rightarrow b$ が最も優れている場合、その適用射 g が *eval* を介して因数分解されるように、他のすべての候補の関数対象を一意にそれにマッピングすることができます。この対象は、私たちのランキングによると、他のどの対象よりも優れています。



普遍関数対象の定義。これは上の図と同じですが、今では対象 $a \Rightarrow b$ が普遍です。

正式には:

a から b への関数対象は、対象 $a \Rightarrow b$ と射

$$eval :: ((a \Rightarrow b) \times a) \rightarrow b$$

であり、それによって他の任意の対象 z には射

$$g :: z \times a \rightarrow b$$

が存在し、それを介して $eval$ を因数分解する唯一の射

$$h :: z \rightarrow (a \Rightarrow b)$$

が存在します。

もちろん、任意の圏の任意の対象のペア a と b に対して、そのような対象 $a \Rightarrow b$ が存在するとは限りません。しかし、**Set** では常に存在します。さらに、**Set** では、この対象はホム集合 $\text{Set}(a, b)$ と同型です。

これが、Haskell では関型 `a -> b` を圏論的な関数対象 $a \Rightarrow b$ として解釈する理由です。

9.2 Curry 化

関数対象の候補をもう一度見てみましょう。しかし、今回は射 g を 2 変数の関数として考えます。

$$g :: z \times a \rightarrow b$$

積からの射であることは、2変数の関数であることに非常に近いです。特に、**Set**では、 g は z の集合と a の集合の値のペアからの関数です。

一方、普遍的な性質は、そのような各 g に対して、 z を関数対象 $a \Rightarrow b$ にマッピングする唯一の射 h が存在することを私たちに教えてくれます。

$$h :: z \rightarrow (a \Rightarrow b)$$

Setでは、これは単に z の型の変数を取り、 a から b への関数を返す関数であることを意味します。これにより、 h は高階関数です。したがって、普遍構成は2変数の関数と1変数で関数を返す関数の間の一対一の対応関係を確立します。この対応関係は *curry* 化と呼ばれ、 h は g の Curry 化されたバージョンと呼ばれます。

この対応関係は一対一です。なぜなら、任意の g に対して一意の h が存在し、任意の h に対して次の式を使って2引数の関数 g を常に再作成できるからです。

$$g = eval \circ (h \times \mathbf{id})$$

関数 g は h の非 *curry* 化バージョンと呼ぶことができます。

Curry 化は本質的に Haskell の構文に組み込まれています。関数を返す関数:

a -> (b -> c)

```
A => (B => C)
```

はしばしば 2 変数の関数として考えられます。それが括弧を使わないシグネチャの読み方です:

```
a -> b -> c
```

```
A => B => C
```

この解釈は、多引数関数を定義する方法で明らかです。例えば:

```
catstr :: String -> String -> String  
catstr s s' = s ++ s'
```

```
val catstr: (String, String) => String =  
(s, s1) => s ++ s1
```

同じ関数は、関数を返す 1 引数の関数 – ラムダとして書くこともできます:

```
catstr' s = \s' -> s ++ s'
```

```
val catstr : String => String => String =  
s => s1 => s ++ s1
```

これらの 2 つの定義は同等であり、いずれも 1 つの引数だけを部分的に適用して、次のような 1 引数の関数を生成することができます:

```
greet :: String -> String
greet = catstr "Hello "

val greet: String => String =
  catstr("Hello ", _)
```

厳密に言えば、2変数の関数とは、ペア(積型)を取る関数です:

$(a, b) \rightarrow c$

$(A, B) \Rightarrow C$

2つの表現間の変換は簡単であり、それを行う2つの(高階)関数は、驚くべきことに、**curry**と**uncurry**と呼ばれます:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f a b = f (a, b)
```

```
def curry[A, B, C](f: (A, B) => C): A => B => C =
  a => b => f(a, b)
```

そして

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (a, b) = f a b
```

```
def uncurry[A, B, C](f: A => B => C): (A, B) => C =  
  (a, b) => f(a)(b)
```

`curry` は、関数対象の普遍構成のための **ファクトライザ**です。これは、次の形式で書き直された場合に特に明らかです：

```
factorizer :: ((a, b) -> c) -> (a -> (b -> c))  
factorizer g = \a -> (\b -> g (a, b))
```

```
def factorizer[A, B, C](g: (A, B) => C): A => (B => C) =  
  a => (b => g(a, b))
```

(リマインダー： ファクトライザは候補から因数分解関数を生成します。）

C++ のような非関数型言語では、Curry 化は可能ですが非自明です。C++ の多引数

関数を Haskell でタプルを取る関数に対応させることができます（ただし、C++ では `std::tuple` を明示的に取る関数を定義できるだけでなく、可変引数関数や初期化リストを取る関数も定義できるため、さらに混乱する可能性があります）。

C++ 関数を部分的に適用するには、テンプレート `std::bind` を使用します。例えば、2 つの文字列を取る関数があるとします：

```
std::string catstr(std::string s1, std::string s2) {  
    return s1 + s2;  
}
```

1つの文字列を取る関数を定義できます:

```
using namespace std::placeholders;  
  
auto greet = std::bind(catstr, "Hello ", _1);  
std::cout << greet("Haskell Curry");
```

Scala は C++ や Java よりも機能的であり、関数を部分適用することができます。予想される場合には、複数の引数リストでそれを定義します:

```
def catstr(s1: String)(s2: String) = s1 + s2
```

もちろん、それはライブラリ作者の先見の明または予知に何らかの量を必要とします。

9.3 指数関数

数学の文献では、関数対象、または 2 つの対象 a と b の間の内部ホム対象はしばしば**指数関数**と呼ばれ、 b^a と表記されます。引数型が指数になっていることに注意してください。この表記は最初は奇妙に思えるかもしれません、関数と積の関係を考えると完全に理にかなっ

ています。私たちは、内部ホム対象の普遍構成を使用するために積を使用する必要がありますが、接続はそれ以上に深いものです。

これは特に有限型間の関数を考えるときに最もよく見られます — **Bool**、**Char**、あるいは **Int** や **Double** のように値の数が有限な型です。そのような関数は、少なくとも原理的には完全にメモ化されたり、ルックアップ用のデータ構造に変換されたりすることができます。そしてこれが関数(射)と関数型(対象)の間の同等性の本質です。

例えば、**Bool** からの(純粋な)関数は完全に 2 つの値で指定されます。1 つは **False** に対応し、もう 1 つは **True** に対応します。任意の **Int** への **Bool** からのすべての可能な関数の集合は、**Int** のペアの集合です。これは、少し創造力を働かせて、**Int** × **Int** または **Int**² と同じです。

別の例として、256 値を含む C++ の型 **char** を見てみましょう (Haskell の **Char** はより大きいです、なぜなら Haskell は Unicode を使用するからです)。C++ 標準ライブラリの一部には、通常ルックアップを使用して実装される関数がいくつかあります。**isupper** や **isspace** のような関数は、256 個のブール値のタプルに相当するテーブルを使用して実装されます。タプルは積型なので、256 個のブール値の積に取り組んでいます: **bool** × **bool** × **bool** × ... × **bool**。算数から知っているように、繰り返される積はべき乗を定義します。256 (または **char**) 回「掛ける」と、**bool** の **char** 乗、つまり **bool**^{**char**} を得ます。

bool の 256 タプル型にはいくつの値がありますか？正確には 2^{256} 個です。これはまた、**char** から **bool** への異なる関数の数です。各関

数は唯一の 256 タプルに対応します。同様に、`bool` から `char` への関数の数は 256^2 であり、以下同様です。関数型の指数記法はこれらの場合に完全に意味を成します。

我々は `int` や `double` からの関数を完全にメモ化したいとは思わないかもしれません。しかし、関数とデータ型の間の等価性は、常に実用的でない場合でも存在します。無限型もあります。例えばリスト、文字列、または木です。それらの型からの関数のイーガーなメモ化は、無限のストレージを必要とします。しかし、Haskell は遅延言語なので、遅延評価される（無限の）データ構造と関数の境界は曖昧です。この関数とデータの二重性は、Haskell の関数型が圏論的な指数対象に対応するという私たちの考え方—つまりデータにより近いもの—と同一視されていることを説明しています。

9.4 デカルト閉圏

型と関数のモデルとして集合の圏を使用し続けるつもりですが、その目的に使用できる圏の大きな族が存在することに触れる価値があります。これらの圏はデカルト閉圏と呼ばれ、`Set` はそのような圏の一例に過ぎません。

デカルト閉圏には以下が含まれる必要があります:

1. 終対象
2. 任意の対象ペアの積

3. 任意の対象ペアの指數関数

指數関数を(無限に多くの回)繰り返し積と考えるならば、任意のアリティの積をサポートする圏としてデカルト閉圏を考えることができます。特に、終対象はゼロの対象の積、または対象のゼロ乗を考えることができます。

コンピュータ科学の視点からデカルト閉圏について興味深いことは、それらが全ての型付けプログラミング言語の基礎を形成する単純型付きラムダ計算のモデルを提供することです。

終対象と積はそれぞれ始対象と余積の双対を持ちます。積が余積上で分配可能である場合

$$\begin{aligned} a \times (b + c) &= a \times b + a \times c \\ (b + c) \times a &= b \times a + c \times a \end{aligned}$$

圏は**双デカルト閉圏**と呼ばれます。私たちは次のセクションで見るよう、**Set**のような双デカルト閉圏はいくつかの興味深い特性を持っています。

9.5 指數関数と代数的データ型

関数型を指數関数と解釈することは代数的データ型のスキームに非常によく適合しています。高校の代数でゼロと一、和、積、指數関数に関連する基本的な恒等性は、それぞれ初期および終対象、余積、積、指數関数について、ほとんど変更されずに双デカルト閉圏の任意のも

ので成立します。私たちはそれらを証明するツールをまだ持っていないません（随伴や米田の補題など）が、それでもここにそれらを列挙して、貴重な直感の源としています。

9.5.1 ゼロ乗

$$a^0 = 1$$

圏的に解釈すると、0 を始対象、1 を終対象に置き換え、等価を同型に置き換えます。指數関数は内部ホム対象です。この特定の指數関数は、始対象から任意の対象 a への射の集合を表します。始対象の定義により、そのような射は正確に 1 つだけ存在するので、ホム集合 $C(0, a)$ は単集合です。単集合は **Set** で終対象なので、この恒等は **Set** で自明に機能します。つまり、それは任意の双デカルト閉圏で機能すると言えます。

Haskell では、0 を **Void** に、1 を **Unit** 型 () に、指數関数を関数型に置き換えます。主張は、**Void** から任意の型 a への関数の集合が **Unit** 型と同等であるということです — つまり、单一要素です。言い換えると、**Void** \rightarrow a のような関数は 1 つだけです。私たちはこの関数を以前に見たことがあります。それは **absurd** と呼ばれています。

これは 2 つの理由で少し厄介です。1 つは、Haskell には本当に無人の型がないということです — すべての型には「終わることのない計算の結果」という底が含まれています。2 つ目の理由は、**absurd** のす

べての実装が同等であり、それらが何をしても、誰もそれらを実行することはできないからです。`absurd` に渡すことができる値はありません。(そして、終わることのない計算を渡すことができたとしても、それは決して返ってこないでしょう！)

9.5.2 1 のべき乗

$$1^a = 1$$

この恒等式は、`Set` で終対象の定義を再述しています: 任意の対象から終対象への一意の射があります。一般的に、 a から終対象への内部ホム対象は、終対象自体と同型です。

Haskell では、任意の型 `a` から `Unit` 型への関数は 1 つだけです。私たちはこの関数を以前に見たことがあります – それは `unit` と呼ばれています。あるいは、`const` を `()` に部分適用した関数と考えることもできます。

9.5.3 第 1 乗

$$a^1 = a$$

これは、終対象からの射が対象 `a` の「要素」を選択するために使用できるという観察の再述です。そのような射の集合は、対象自体と同型

です。Set では、および Haskell では、同型は集合 a の要素とそれらの要素を選択する関数、 $(\lambda) \rightarrow a$ の間のものです。

9.5.4 和の指數関数

$$a^{b+c} = a^b \times a^c$$

圈的には、2つの対象の余積からの指數関数が、2つの指數関数の積と同型であることを示しています。Haskell では、この代数的恒等式は非常に実用的な解釈を持っています。それは、2つの型の和からの関数が、個々の型からの一対の関数と等価であることを教えてくれます。これは私たちが和で関数を定義するときに使用するケース分析です。1つの関数定義を `case` 文で書く代わりに、通常はそれを 2つ(またはそれ以上)の関数に分割します。それぞれが型構造から個別に処理します。例えば、(`Either Int Double`) の和型からの関数を取ります:

```
f :: Either Int Double -> String
```

```
val f: Either[Int, Double] => String
```

それは、それぞれ `Int` と `Double` からの一対の関数として定義することができます:

```
f (Left n) = if n < 0 then "Negative int" else "Positive int"
f (Right x) = if x < 0.0 then "Negative double" else "Positive double"

val f: Either[Int, Double] => String = {
  case Left(n) => if (n < 0) "Negative int" else "Positive int"
  case Right(x) => if (x < 0.0) "Negative double" else "Positive double"
}
```

ここで、`n` は `Int` であり、`x` は `Double` です。

9.5.5 指数関数の指数関数

$$(a^b)^c = a^{b \times c}$$

これは、純粋に指数対象の用語で Curry 化を表現する方法です。関数を返す関数は、積(2引数の関数)からの関数と同等です。

9.5.6 積の上の指数関数

$$(a \times b)^c = a^c \times b^c$$

Haskell では: ペアを返す関数は、ペアの各要素を生成する一対の関数と同等です。

これらの簡単な高校の代数的恒等式が圏論に持ち上げられ、関数型プログラミングの実践的な応用を持っているのはかなり信じられないことです。

9.6 Curry-Howard 同型

私はすでに論理と代数的データ型の間の対応について言及しました。`Void` 型と `Unit` 型 `()` は偽と真に対応します。積型と和型は、それぞれ論理的な接続 \wedge (AND) と選言 \vee (OR) に対応します。このスキームで、私たちが定義した関数型は論理的な含意 \Rightarrow に対応します。言い換えると、型 $a \rightarrow b$ は「 a ならば b 」と読むことができます。

Curry-Howard 同型によれば、すべての型は命題として解釈できます – 真または偽かもしれない宣言または判断です。そのような命題は、型が居住している場合は真と見なされ、そうでない場合は偽と見なされます。特に、論理的な含意は、それに対応する関数型が居住している場合に真であり、それはその型の関数が存在することを意味します。関数の実装はしたがって定理の証明です。プログラムを書くことは定理を証明することに等しいです。いくつかの例を見てみましょう。

関数対象の定義で導入した関数 `eval` を取ります。そのシグネチャは:

```
eval :: ((a → b), a) → b
```

```
def eval[A, B]: ((A => B), A) => B
```

それは関数とその引数からなるペアを取り、適切な型の結果を生成します。それは射:

$$eval :: (a \Rightarrow b) \times a \rightarrow b$$

を Haskell で実装したものです。これは関数型 $a \Rightarrow b$ (または指数対象 b^a) を定義します。このシグネチャを Curry-Howard 同型を使って論理的な述語に変換しましょう:

$$((a \Rightarrow b) \wedge a) \Rightarrow b$$

この文は次のように読むことができます: b が a から導かれることが真であり、かつ a が真であるならば、 b も真でなければなりません。これは直感的に非常に理にかなっており、古代から *modus ponens* として知られています。この定理を実装することで証明します:

```
eval :: ((a -> b), a) -> b
eval (f, x) = f x
```

```
def eval[A, B]: ((A => B), A) => B =
(f, x) => f(x)
```

あなたが関数 f (a を取って b を返す) と、型 a の具体的な値 x からなるペアを私に与えれば、私は単に関数 f を x に適用することで、型 b

の具体的な値を生成することができます。この関数を実装することで、 $((a \rightarrow b), a) \rightarrow b$ の型が居住されていることを示しました。したがって、*modus ponens* は私たちの論理で真です。

では、明らかに偽の述語はどうでしょうか？ 例えば: a または b が真であれば a も真でなければなりません。

$$a \vee b \Rightarrow a$$

これは明らかに間違っています。なぜなら、偽の a と真の b を選ぶことができるからです。それは反例です。

Curry-Howard 同型を使用してこの述語を関数シグネチャにマッピングすると、次のようになります:

`Either a b -> a`

`Either[A, B] => A`

いくら試みても、この関数を実装することはできません—**Right** 値で呼ばれた場合、**a** 型の値を生成することはできません。(純粋関数について話していることを忘れないでください。)

最後に、**absurd** 関数の意味について考えてみましょう:

`absurd :: Void -> a`

```
def absurd[A]: Nothing => A
```

`Void` が偽に対応することを考えると、次のようにになります:

$$\text{false} \Rightarrow a$$

偽からは何でも導き出される (*ex falso quodlibet*)。以下は、Haskell でのこの文 (関数) の 1 つの可能な証明 (実装) です:

```
absurd (Void a) = absurd a
```

ここで `Void` は次のように定義されています:

```
newtype Void = Void Void
```

常に、`Void` 型は厄介です。この定義により、値を構成することは不可能になります。なぜなら、1 つを提供するためには、すでに 1 つを持っている必要があるからです。したがって、関数 `absurd` は決して呼び出すことができません。

これらはすべて興味深い例ですが、Curry-Howard 同型には実用的な側面があるのでしょうか？おそらく日常のプログラミングではありません。しかし、Agda や Coq のようなプログラミング言語は Curry-Howard 同型を利用して定理を証明します。

コンピュータは数学者が仕事をするのを助けるだけでなく、数学の非常に基礎を革命しています。その分野での最新の熱い研究トピックはホモトピー型理論と呼ばれ、型理論の成長です。それは Booleans、整数、積と余積、関数型などでいっぱいです。そして、疑問を払拭す

るかのように、この理論は Coq や Agda で定式化されています。コンピュータは複数の方法で世界を革命しています。

9.7 参考文献

1. Ralph Hinze, Daniel W. H. James, **Reason Isomorphically!**^{*1}. この論文には、この章で私が言及した高校の代数的恒等式のすべての証明が含まれています。

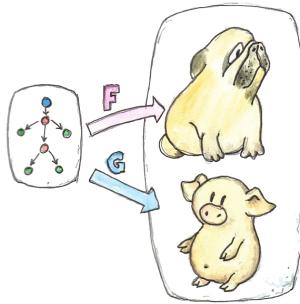
^{*1} <http://www.cs.ox.ac.uk/ralf.hinze/publications/WGP10.pdf>

10

自然変換

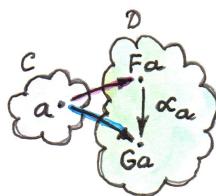
関手について、それが圏間の構造を保つ写像として機能することを話しました。

関手はある圏を別の圏に「埋め込む」ものです。それは多くのものを一つにまとめることはありますが、接続を断つことはありません。関手を使って、ある圏を別の圏内でモデル化していると考えることができます。ソース圏は、ターゲット圏の一部分の構造のためのモデル、設計図として機能します。



ある圏を別の圏に埋め込む方法は多くあります。時にはそれらは同等のものもあれば、全く異なるものもあります。一つはソース圏全体を一つの対象に収縮させるかもしれません、別のものは各対象を異なる対象に、各射を異なる射に写像するかもしれません。同じ設計図が多く異なる方法で実現されるのです。自然変換はこれらの実現を比較するのに役立ちます。関手の写像です—それらの関手的性質を保持する特別な写像です。

圏 C と D の間の関手 F と G を考えてください。 C のただ一つの対象 a に注目すると、それは二つの対象、 Fa と Ga に写像されます。したがって、関手の写像は Fa を Ga に写像するべきです。



Fa と Ga は同じ圏 \mathbf{D} の対象であることに注意してください。同じ圏の対象間の写像は、圏の性質に逆らってはなりません。我々は対象間に人工的な接続を作りたくありません。したがって、既存の接続、すなわち射を使用するのが**自然**です。自然変換は射の選択です: 各対象 a に対して、 Fa から Ga への一つの射を選びます。自然変換を α と呼ぶと、この射は α の a における**コンポーネント**、または α_a と呼ばれます。

$$\alpha_a :: Fa \rightarrow Ga$$

a は \mathbf{C} の対象であるのに対し、 α_a は \mathbf{D} の射であることを念頭に置いてください。

もし a について、 Fa と Ga の間に \mathbf{D} に射が存在しない場合、 F と G の間に自然変換は存在しません。

もちろん、これは話の半分だけです。なぜなら関手は対象だけでなく射も写像するからです。では、自然変換はそれらの写像とはどのように関わるのでしょうか? 射の写像は固定されています — F と G の間の任意の自然変換の下で、 Ff は Gf に変換されなければなりません。さらに、二つの関手による射の写像は、それと整合する自然変換を定義する際の選択肢を大きく制限します。 \mathbf{C} の二つの対象 a と b の間の射 f を考えてみましょう。それは二つの射、 Ff と Gf に \mathbf{D} で写像されます:

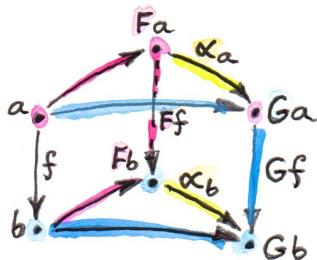
$$Ff :: Fa \rightarrow Fb$$

$$Gf :: Ga \rightarrow Gb$$

自然変換 α は、 D で図式を完成させる二つの追加の射を提供します：

$$\alpha_a :: Fa \rightarrow Ga$$

$$\alpha_b :: Fb \rightarrow Gb$$

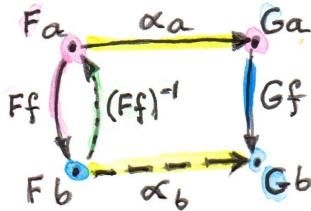


これで Fa から Gb への二つの方法があります。それらが等しいことを確かめるために、任意の f に対して成り立つ**自然性条件**を課さなければなりません：

$$Gf \circ \alpha_a = \alpha_b \circ Ff$$

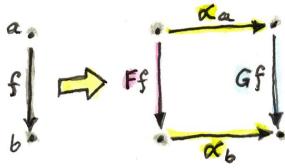
自然性条件はかなり厳しい要求です。例えば、もし射 Ff が可逆なら、自然性は α_b を α_a の項で決定します。それは f に沿って α_a を輸送します：

$$\alpha_b = (Gf) \circ \alpha_a \circ (Ff)^{-1}$$



もし二つの対象の間に一つ以上の可逆射が存在するなら、これらの輸送は全て一致しなければなりません。一般に、射は可逆でないことが多いですが、二つの関手間に自然変換が存在する可能性は決して保証されていません。そのため、自然変換によって関連付けられる関手の希少性や豊富さは、それらが操作する圏の構造について多くのことを教えてくれるかもしれません。極限や米田の補題について話をするときに、その例をいくつか見ることになります。

自然変換をコンポーネントごとに見ると、それは対象を射に写像すると言えます。自然性条件のために、それは射を可換な四角形に写像するとも言えます – **C** の各射について **D** に一つの可換な自然性の四角形があります。



自然変換のこの性質は、多くの圏論的構成に非常に便利で、それらはしばしば可換図式を含んでいます。関手の賢明な選択によって、多くのこれらの可換性条件は自然性条件に変換することができるかもしれません。極限、余極限、そして随伴について話をするときに、その例をいくつか見ることになります。

最後に、自然変換は関手の同型を定義するために使用することができます。二つの関手が自然に同型であると言うことは、ほとんど同じであると言うようなものです。自然同型は、そのコンポーネントが全て同型(可逆射)である自然変換として定義されます。

10.1 多相的関数

プログラミングにおける関手(具体的には、自己関手)の役割について話しました。それらは型を型に写像する型構造に相当します。関手は関数も関数に写像しますが、この写像は `fmap`(または C++ での `transform`、`then` など) という高階関数によって実装されます。

自然変換を構成するには、ここでは型、`a` の対象から始めます。ある関手、`F` はそれを型 Fa に写像します。別の関手、`G` はそれを Ga に写像します。自然変換 `alpha` の `a` におけるコンポーネントは、 Fa から Ga への関数です。擬似 Haskell で：

```
alphaa :: F a -> G a
```

自然変換は、全ての型 `a` に対して定義される多相的関数です：

```
alpha :: forall a . F a -> G a
```

```
def alpha[A]: F[A] => G[A]
```

`forall a` は Haskell ではオプションです (実際には、言語拡張 `ExplicitForAll` をオンにする必要があります)。通常、このように書きます：

```
alpha :: F a -> G a
```

```
val alpha: F[A] => G[A]
```

心に留めておくべきことは、これは実際には `a` によってパラメータ化された関数の族であるということです。これは Haskell 構文の簡潔さの別の例です。C++ での同様の構造は少し冗長になるでしょう：

```
template<class A> G<A> alpha(F<A>);
```

Haskell の多相的関数と C++ のジェネリック関数の間には、これらの関数が実装され、型検査される方法に反映されているより深い違いがあります。Haskell では、多相的関数はすべての型に対して一様に定義されなければなりません。一つの公式がすべての型にわたって機能しなければなりません。これは**パラメトリック多相性**と呼ばれます。

一方、C++ はデフォルトで**アドホック多相性**をサポートしています。これは、テンプレートがすべての型に対して正しく定義される必要がないことを意味します。テンプレートが与えられた型で機能するかどうかは、具体的な型が型パラメータに代入されるインスタンス化の時に決定されます。型チェックは遅延され、残念ながらしばしば理解しがたいエラーメッセージを引き起こします。

C++ にはまた、異なる型に対して同じ関数の異なる定義を可能にする関数のオーバーロードとテンプレートの特殊化のメカニズムもあります。Haskell では、この機能は型クラスと型の族によって提供されます。

Haskell のパラメトリック多相性は意外な結果を持ちます: 型が:

```
alpha :: F a -> G a
```

```
val alpha: F[A] => G[A]
```

ここで、 F と G は関手で、任意の多相的関数は自然性条件を自動的に満たします。ここで、圏論的な記法で (f は関数 $f :: a \rightarrow b$ です) :

$$Gf \circ \alpha_a = \alpha_b \circ Ff$$

Haskell では、関手 G 上の射 f の作用は `fmap` を使用して実装されます。まず、明示的な型注釈を使って擬似 Haskell で書きます:

```
fmap_G f . alpha_a = alpha_b . fmap_F f
```

型推論のため、これらの注釈は必要ありません、そして次の方程式が成立します:

```
fmap f . alpha = alpha . fmap f
```

これはまだ本物の Haskell ではありません – 関数の等価性はコードで表現できないからです – しかし、それはプログラマが等式推論に使用したり、コンパイラが最適化を実装するために使用できる恒等式です。

Haskell における自然性条件の自動性は「無料の定理」と関連しています。Haskell で自然変換を定義するために使用されるパラメトリック多相性は、すべての型にわたって一つの公式を課すという非常に強い制約を課します。これらの制約は、そのような関数に関する等式定

理に変換されます。関手を変換する関数の場合、無料の定理は自然性条件です。^{*1}

関手を Haskell で考える一つの方法は、それらを一般化されたコンテナと見なすことです。この類推を続けて、自然変換を考えることができます。それは一つのコンテナの内容を別のコンテナに再梱包するためのレシピです。我々はアイテム自体に触れてはいけません: 私たちはそれらを変更したり、新しいものを作成したりしてはいけません。私たちはただ、それらを新しいコンテナに(時には複数回) コピーするだけです。

自然性条件は、アイテムを最初に `fmap` を通して変更し、後で再梱包するか、あるいは最初に再梱包してから、新しいコンテナの独自の実装でアイテムを変更するか、どちらを行っても問題がないという声明です。これら二つのアクション、再梱包と `fmap` は直交しています。「一方は卵を動かし、もう一方はそれらを茹でます。」

Haskell での自然変換のいくつかの例を見てみましょう。最初は、リスト関手と `Maybe` 関手の間です。リストが空でない場合に限り、リストの先頭を返します:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

^{*1} 無料の定理についてもっと読むことができますブログ “Parametricity: Money for Nothing and Theorems for Free.”

```
def safeHead[A]: List[A] => Option[A] = {  
    case Nil => None  
    case x :: xs => Some(x)  
}
```

これは `a` に多相的な関数です。それはどんな型 `a` に対しても制限なく機能するので、パラメトリック多相性の一例です。したがって、それは二つの関手間の自然変換です。しかし、自然性条件を確認するために、私たち自身を納得させるために、それを検証しましょう。

```
fmap f . safeHead = safeHead . fmap f  
  
(fmap(f) compose safeHead) == (safeHead compose fmap(f))
```

考慮すべき二つのケースがあります。空リスト:

```
fmap f (safeHead []) = fmap f Nothing = Nothing  
  
fmap(f)(safeHead(List.empty)) == fmap(f)(None) == None  
  
safeHead (fmap f []) = safeHead [] = Nothing  
  
safeHead(fmap(f)(List.empty)) == safeHead(List.empty) == None
```

そして空でないリスト:

```
fmap f (safeHead (x:xs)) = fmap f (Just x) = Just (f x)
```

```
fmap(f)(safeHead(x :: xs)) == fmap(f)(Some(x)) == Some(f(x))
```

```
safeHead (fmap f (x:xs)) = safeHead (f x : fmap f xs) = Just (f x)
```

```
safeHead(fmap(f)(x :: xs)) == safeHead(f(1) :: fmap(f)(xs)) == Some(f(x))
```

私はリストのための `fmap` の実装を使用しました:

```
fmap f [] = []
fmap f (x:xs) = f x : fmap f xs
```

```
def fmap[A, B](f: A => B): List[A] => List[B] = {
    case Nil => Nil
    case x :: xs => f(x) :: fmap(f)(xs)
}
```

そして `Maybe` のための:

```
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
    case None => None
    case Some(x) => Some(f(x))
```

```
}
```

興味深いケースは、関手の一つが自明な **Const** 関手である場合です。**Const** 関手からまたはそれへの自然変換は、それが返り型で多相的であるか、引数型で多相的であるかのいずれかのように見える関数のようです。

例えば、**length** はリスト関手から **Const Int** 関手への自然変換として考えることができます:

```
length :: [a] -> Const Int a
length [] = Const 0
length (x:xs) = Const (1 + unConst (length xs))
```

```
def length[A]: List[A] => Const[Int, A] = {
  case Nil => Const(0)
  case x :: xs => Const(1 + unConst(length(xs)))
}
```

ここで、**unConst** は **Const** コンストラクタを剥がすために使用されます:

```
unConst :: Const c a -> c
unConst (Const x) = x

def unConst[C, A]: Const[C, A] => C = {
  case Const(x) => x
}
```

もちろん、実際には `length` は次のように定義されています:

```
length :: [a] -> Int  
  
def length[A]: List[A] => Int
```

これは実際にそれが自然変換であるという事実を隠しています。

`Const` 関手からのパラメトリック多相的関数を見つけることは少し難しいかもしれません。それは何もないところから値を作り出す必要があるためです。最善を尽くせば:

```
scam :: Const Int a -> Maybe a  
scam (Const x) = Nothing  
  
def scam[A]: Const[Int, A] => Option[A] = {  
  case Const(x) => None  
}
```

私たちがすでに見たもう一つの一般的な関手は、`Reader` 関手です。私はその定義を `newtype` として書き直します:

```
newtype Reader e a = Reader (e -> a)  
  
case class Reader[E, A](run: E => A)
```

それは二つの型によってパラメータ化されていますが、二つ目のものにのみ(共変的に)関手的です:

```
instance Functor (Reader e) where
    fmap f (Reader g) = Reader (\x -> f (g x))

implicit def readerFunctor[E] = new Functor[Reader[E, ?]] {
    def fmap[A, B](f: A => B)(g: Reader[E, A]): Reader[E, B] =
        Reader(x => f(g.run(x)))
}
```

あなたは任意の型 `e` に対して、`Reader e` から任意の他の関手 `f` への自然変換の族を定義することができます。後で見るように、この族のメンバーは常に `f e` の要素と一一対応関係にあります(米田の補題)。

例えば、要素 `()` を持つ `Unit` 型 `()` を考えてみましょう。関手 `Reader ()` は任意の型 `a` を関数型 `() -> a` に写像します。これらは単に `a` の集合から单一の要素を選ぶすべての関数です。`a` の要素と同じ数だけこれらが存在します。ここで、この関手から `Maybe` 関手への自然変換を考えてみましょう:

```
alpha :: Reader () a -> Maybe a

def alpha[A]: Reader[Unit, A] => Option[A]
```

これらのうちの二つだけが、`dumb` と `obvious` です：

```
dumb (Reader _) = Nothing

def dumb[A]: Reader[Unit, A] => Option[A] = {
    case Reader(_) => None
}
```

そして

```
obvious (Reader g) = Just (g())

def obvious[A]: Reader[Unit, A] => Option[A] = {
    case Reader(g) => Some(g())
}
```

(`g` ができる唯一のことは、それを単位値 `()` に適用することです。)

そして確かに、米田の補題によって予測されるように、これらは `Maybe ()` 型の二つの要素、`Nothing` と `Just ()` に対応しています。後で米田の補題に戻ります — これはちょっとした予告でした。

10.2 自然性を越えて

二つの関手 (`Const` 関手のエッジケースを含む) の間のパラメトリック多相的関数は常に自然変換です。すべての標準的な代数的データ型が関手であるため、そのような型間の多相的関数は自然変換です。

また、関数型を私たちの手元に持っています。これらは返り値の型で関手的です。私たちはそれらを関手 (**Reader** 関手のような) を構成し、高階関数である自然変換を定義するために使用することができます。

ただし、関数型は引数型で共変ではありません。反変です。もちろん反変関手は、逆圏からの共変関手に相当します。二つの反変関手間の多相的関数はまだ圏論的な意味で自然変換ですが、それは逆圏から Haskell 型への関手に対して機能します。

以前に見た反変関手の例を覚えているかもしれません:

```
newtype Op r a = Op (a -> r)

case class Op[R, A](f: A => R)
```

この関手は `a` で反変です:

```
instance Contravariant (Op r) where
    contramap f (Op g) = Op (g . f)

implicit def opContravariant[R] = new Contravariant[Op[R, ?]] {
    def contramap[A, B](f: B => A): Op[R, A] => Op[R, B] = {
        case Op(g) => Op(g compose f)
    }
}
```

例えば、`Op Bool` から `Op String` への多相的関数を書くことができます:

```
predToStr (Op f) = Op (\x -> if f x then "T" else "F")
```

```
def predToStr[A]: Op[Boolean, A] => Op[String, A] = {
  case Op(f) => Op(x => if (f(x)) "T" else "F")
}
```

しかし、二つの関手が共変でないため、これは `Hask` での自然変換ではありません。ただし、それらが共に反変であるため、「逆」の自然性条件を満たします:

```
contramap f . predToStr = predToStr . contramap f
```

```
(op.contramap(func) compose predToStr) == (predToStr compose
  ↵ op.contramap(func))
```

`f` は、`fmap` を使用する際に使用するものとは逆方向でなければならぬことに注意してください。これは `contramap` の署名からです:

```
contramap :: (b -> a) -> (Op Bool a -> Op Bool b)
```

```
def contramap[A, B](f: B => A): Op[Boolean, A] => Op[Boolean, B] = {
  case Op(g) => Op(g compose f)
```

}

共変でも反変でもない型コンストラクタがありますか？ ここに一つの例があります：

`a -> a`

`A => A`

これは関手ではありません。なぜなら同じ型 `a` が負(反変)と正(共変)の位置の両方で使用されているためです。そのため、`fmap` や `contramap` をこの型のために実装することはできません。したがって、以下の署名を持つ関数：

`(a -> a) -> f a`

`(A => A) => F[A]`

ここで `f` は任意の関手で、自然変換にはなり得ません。興味深いことに、自然変換の一般化である、双自然変換と呼ばれるものが、このようなケースを扱います。私たちは、エンドについて議論する時に、それらについて説明します。

10.3 関手圏

関手間の写像 – 自然変換 – があると、関手が圏を形成するかという疑問が自然と生じます。そして実際、それらはします！ 圏のペアごとに一つの関手圏があります。この圏の対象は **C** から **D** への関手で、射はそれらの関手間の自然変換です。

二つの自然変換の合成を定義する必要がありますが、それはかなり簡単です。自然変換のコンポーネントは射であり、射の合成方法を知っています。

実際、関手 F から G への自然変換 α を取りましょう。対象 a におけるそのコンポーネントは何らかの射です：

$$\alpha_a :: Fa \rightarrow Ga$$

関手 G から H への自然変換 β と α を合成したいと思います。 a における β のコンポーネントは射です：

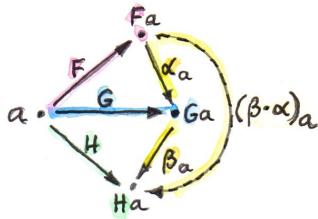
$$\beta_a :: Ga \rightarrow Ha$$

これらの射は合成可能で、その合成は別の射です：

$$\beta_a \circ \alpha_a :: Fa \rightarrow Ha$$

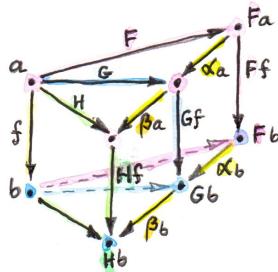
この射を自然変換 $\beta \cdot \alpha$ のコンポーネントとして使用します – 二つの自然変換 β の後に α の合成です：

$$(\beta \cdot \alpha)_a = \beta_a \circ \alpha_a$$



一つ(長い)図を見るだけで、この合成が実際には F から H への自然変換であることが確信できます:

$$Hf \circ (\beta \cdot \alpha)_a = (\beta \cdot \alpha)_b \circ Ff$$



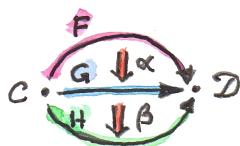
自然変換の合成は結合的です、なぜならそれらのコンポーネント、通常の射であり、その合成に関して結合的だからです。

最後に、各関手 F には、そのコンポーネントが恒等射である自己恒等自然変換 1_F があります:

$$\mathbf{id}_{Fa} :: Fa \rightarrow Fa$$

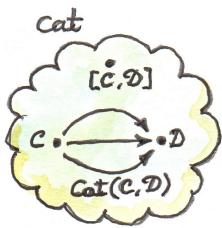
したがって、確かに、関手は圏を形成します。

表記法について一言。Saunders Mac Lane に従って、私は自然変換合成の種類について説明したばかりのものにドットを使用します。問題は、自然変換を合成する二つの方法があるということです。この一つは垂直合成と呼ばれます、なぜなら関手は通常、それを説明する図で縦に積み上げられるからです。垂直合成は関手圏を定義するのに重要です。間もなく水平合成について説明します。



圏 C と D の間の関手圏は $\text{Fun}(C, D)$ 、または $[C, D]$ 、時には D^C と書かれます。この最後の表記は、関手圏自体が何らかの他の圏の関数対象(指数関数)と見なされうることを示唆しています。これは実際にそういうのでしょうか？

これまでに構成してきた抽象化の階層を見てみましょう。我々は対象と射の集まりである圏から始めました。圏自体(厳密には、対象が集合を形成する小さい圏)は、より高次元の圏 Cat の対象です。その圏の射は関手です。 Cat のホム集合は関手の集合です。例えば $\text{Cat}(C, D)$ は圏 C と D の間の関手の集合です。



関手圏 $[C, D]$ もまた二つの圏間の関手の集合 (及び射としての自然変換) です。その対象は $\text{Cat}(C, D)$ のメンバーと同じです。さらに、関手圏は、圏であるため、それ自体が Cat の対象でなければなりません (二つの小さい圏間の関手圏はそれ自体小さいです)。我々は圏内のホム集合と同じ圏内の対象との間の関係を持っています。この状況は、前章で見た指數関数対象と全く同じです。 Cat で後者をどのように構成するか見てみましょう。

指數関数を構成するためには、まず積を定義する必要があります。 Cat では、これは比較的簡単です。なぜなら小さい圏は**集合**の対象であり、集合のデカルト積を定義する方法を知っているからです。そのため、積圏 $C \times D$ の対象は単に対象のペア、 (c, d) で、一つは C から、もう一つは D からです。同様に、二つのそのようなペア、 (c, d) と (c', d') の間の射は、射のペア、 (f, g) で、 $f :: c \rightarrow c'$ と $g :: d \rightarrow d'$ です。これらの射のペアはコンポーネントごとに合成され、常に恒等射のペアが存在します。それは単に恒等射のペアです。要するに、 Cat はデカルト閉圏であり、そこには任意の圏のペアの指數関数対象があ

ります。そして「対象」によって **Cat** で意味するのは圏なので、 D^C は圏です。これを圏 **C** と **D** の間の関手圏と同一視することができます。

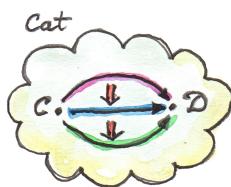
10.4 2-圏

それで済んだとして、**Cat** をより詳細に見てみましょう。定義により、**Cat** の任意のホム集合は関手の集合です。しかし、我々が見てきたように、二つの対象間の関手は単なる集合以上の豊かな構造を持っています。それらは自然変換を射として持つ圏を形成します。関手が **Cat** の射と見なされる場合、自然変換は射間の射です。

この豊かな構造は、2-圏の一例です。これは、対象と射（この文脈では1-射と呼ばれるかもしれません）の他に、射間の射、つまり2-射がある圏の一般化です。

Cat を2-圏と見なした場合、我々は持っています:

- 対象: (小さい) 圏
- 1-射: 圏間の関手
- 2-射: 関手間の自然変換。



圏 \mathbf{C} と \mathbf{D} の間のホム集合の代わりに、我々はホム圏、関手圏 $\mathbf{D}^{\mathbf{C}}$ を持っています。我々は通常の関手合成を持っています: 圏 $\mathbf{D}^{\mathbf{C}}$ からの関手 F は、圏 $\mathbf{E}^{\mathbf{D}}$ からの関手 G と合成して $\mathbf{E}^{\mathbf{C}}$ からの $G \circ F$ を与えます。しかし、我々はまた、各ホム圏内の合成を持っています – 関手、または 2-射間の自然変換の垂直合成。

二つの合成がある 2-圏では、それらがどのように相互作用するかという問題が生じます。

\mathbf{Cat} の二つの関手、または 1-射を選びましょう:

$$F :: \mathbf{C} \rightarrow \mathbf{D}$$

$$G :: \mathbf{D} \rightarrow \mathbf{E}$$

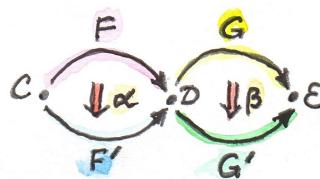
そしてそれらの合成:

$$G \circ F :: \mathbf{C} \rightarrow \mathbf{E}$$

関手 F と G に作用する二つの自然変換、 α と β を持っていると仮定しましょう:

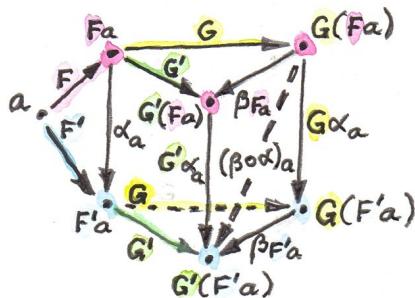
$$\alpha :: F \rightarrow F'$$

$$\beta :: G \rightarrow G'$$



α のターゲットは β のソースと異なるため、このペアに垂直合成を適用することはできません。実際、それらは二つの異なる関手圏のメンバーです: D^C と E^D 。しかし、 F' と G' の関手に合成を適用することはできます。なぜなら F' のターゲットは G' のソース、つまり圏 D です。 $G' \circ F'$ と $G \circ F$ の関係は何でしょうか？

α と β を持っている場合、 $G \circ F$ から $G' \circ F'$ への自然変換を定義できますか？構成をスケッチします。



通常のように、圏 C の対象 a から始めます。その像は D の二つの対象、 Fa と $F'a$ に分裂します。また、 α のコンポーネントであるこれら二つの対象をつなぐ射があります:

$$\alpha_a :: Fa \rightarrow F'a$$

D から E へ行くと、これら二つの対象はさらに四つの対象に分裂します: $G(Fa)$, $G'(Fa)$, $G(F'a)$, $G'(F'a)$ 。また、四つの射を形成する四角形もあります。これらの射のうち二つは自然変換 β のコンポーネント

です:

$$\begin{aligned}\beta_{Fa} &:: G(Fa) \rightarrow G'(Fa) \\ \beta_{F'a} &:: G(F'a) \rightarrow G'(F'a)\end{aligned}$$

他の二つは、関手による α_a の像です(関手は射を写像します):

$$\begin{aligned}G\alpha_a &:: G(Fa) \rightarrow G(F'a) \\ G'\alpha_a &:: G'(Fa) \rightarrow G'(F'a)\end{aligned}$$

多くの射があります。私たちの目標は、 $G(Fa)$ から $G'(F'a)$ への射を見つけることです。これは $G \circ F$ と $G' \circ F'$ の二つの関手をつなぐ自然変換のコンポーネントの候補です。実際には $G(Fa)$ から $G'(F'a)$ への二つのパスがあります:

$$\begin{aligned}G'\alpha_a \circ \beta_{Fa} \\ \beta_{F'a} \circ G\alpha_a\end{aligned}$$

幸いにも、これらは等しいです。なぜなら、我々が形成した四角形は β の自然性の四角形であるためです。

これで、 $G \circ F$ から $G' \circ F'$ への自然変換のコンポーネントを定義しました。この変換の自然性の証明は、十分な忍耐があれば比較的簡単です。

この自然変換を α と β の**水平合成**と呼びます:

$$\beta \circ \alpha :: G \circ F \rightarrow G' \circ F'$$

再び、Mac Lane に従って水平合成のために小さい円を使用しますが、その代わりに星を使用することもあります。

ここで圏論的な経験則があります: 合成があるたびに、その圏を探すべきです。自然変換の垂直合成は、関手圏の一部です。しかし、水平合成についてはどうでしょうか？ それはどの圏に属していますか？

これを理解する方法は、**Cat** を横から見ることです。自然変換を関手間の射としてではなく、圏間の射として考えます。自然変換は二つの圏、関手がつなぐそれらを間に置いています。私たちはそれをこれら二つの圏をつなぐものとして考えることができます。



Cat の二つの対象、圏 C と D に焦点を当てましょう。 C につながる関手間の自然変換の集合があります。これらの自然変換は、 C から D への我々の新しい射です。同じように、 D から E へつながる関手間の自然変換があります。これらを D から E への新しい射として扱います。水平合成は、これらの射の合成です。

また、 C から C への恒等射があります。それは C 上の恒等関手に自分自身をマッピングする恒等自然変換です。水平合成の恒等射は、垂直合成の恒等射でもありますが、逆は必ずしも真ではありません。

最後に、二つの合成は交換則を満たします:

$$(\beta' \cdot \alpha') \circ (\beta \cdot \alpha) = (\beta' \circ \beta) \cdot (\alpha' \circ \alpha)$$

ここで、Saunders Mac Lane の言葉を引用します: この事実を証明するために必要な明白な図式を書き下すことを楽しむ読者がいるかもしれません。

さらに、将来的に役立つかもしれないもう一つの表記法があります。この新しい横向きの解釈の **Cat** で、対象から対象への二つの方法があります: 関手を使用するか、自然変換を使用するかです。しかし、我々は関手の矢印をこの関手に作用する恒等自然変換として特別な種類の自然変換として再解釈することができます。そのため、このような表記をよく見かけます:

$$F \circ \alpha$$

ここで F は圏 **D** から **E** への関手であり、 α は圏 **C** から **D** への二つの関手間の自然変換です。関手と自然変換を合成することはできませんが、これは恒等自然変換 1_F の後に α を水平合成すると解釈されます。

同様に:

$$\alpha \circ F$$

は α の後に 1_F を水平合成すると解釈されます。

10.5 結論

これで本書の第一部が終わります。私たちは圏論の基本的な語彙を学びました。対象と圏を名詞として、射、関手、自然変換を動詞として考えることができます。射は対象をつなぎ、関手は圏をつなぎ、自然変換は関手をつなぎます。

しかし、我々はまた見てきましたが、あるレベルの抽象化での行為は、次のレベルでの対象となります。射の集合は関数対象に変わります。対象として、それは他の射のソースまたはターゲットになることができます。これが高階関数の背後にあるアイディアです。

関手は対象を対象に写像するので、私たちはそれを型コンストラクタまたはパラメトリック型として使用することができます。関手はまた射を写像するので、それは高階関数 – **fmap** です。いくつかのシンプルな関手、例えば **Const**、積、余積は、多様な代数的データ型を生成するために使用することができます。関数型もまた関手的です、共変的にも反変的にも、そして代数的データ型を拡張するために使用することができます。

関手は、関手圏の対象と見なすことができます。そのようにして、彼らは射のソースとターゲットになります: 自然変換。自然変換は、特別なタイプの多相的関数です。

10.6 チャレンジ

1. **Maybe** 関手からリスト関手への自然変換を定義し、それに対する自然性条件を証明してください。
2. **Reader ()** とリスト関手の間で少なくとも二つの異なる自然変換を定義してください。いくつ異なるリストの () が存在しますか？
3. 前の課題を **Reader Bool** と **Maybe** で続けてください。

4. 自然変換の水平合成が自然性条件を満たすことを示してください (ヒント: コンポーネントを使用します)。図式を追う良い練習です。
5. 交換則を証明するために必要な明白な図式を書き下すことを樂しむかどうかについて短いエッセイを書いてください。
6. 異なる `Op` 関手間の変換の逆自然性条件のいくつかのテストケースを作成してください。一つの選択は以下の通りです:

```
op :: Op Bool Int  
op = Op (\x -> x > 0)
```

そして

```
f :: String -> Int  
f x = read x
```

第 2 部

11

宣言的プログラミング

この本の最初の部分で、圏論とプログラミングが共に合成可能性についてであると主張しました。プログラミングでは、問題を処理可能なレベルになるまで分解し、それぞれの部分問題を順番に解決し、解決策をボトムアップで再合成します。大まかに言えば、これには二つの方法があります: コンピュータに何をするかを伝える方法と、どのようにそれをするかを伝える方法です。前者は宣言的プログラミングと呼ばれ、後者は命令的です。

これは最も基本的なレベルでも見ることができます。合成自体は、宣言的に定義される場合があります。つまり、 h は f の後に g の合成です:

```
h = g . f
```

```
val h = g compose f
```

または命令的に；つまり、最初に `f` を呼び出し、その呼び出しの結果を記憶し、その結果を使って `g` を呼び出します：

```
h x = let y = f x  
      in g y
```

```
val h = x => {  
  val y = f(x)  
  g(y)  
}
```

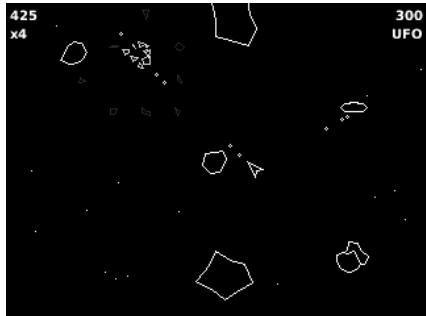
プログラムの命令的バージョンは通常、時間順序で並べられた一連のアクションとして記述されます。特に、`f` の実行が完了する前に `g` を呼び出すことはできません。少なくとも、それが概念的な絵です—遅延言語では、必要に応じて呼び出し引数渡しで、実際の実行は異なる方法で進行するかもしれません。

実際には、コンパイラの賢さに応じて、宣言的コードと命令的コードの実行方法にはあまり違いがないかもしれません。しかし、問題解決へのアプローチ方法と、結果としてのコードの保守性およびテスト可能性の点で、二つの方法論は時に劇的に異なります。

主な問題は、問題に直面したとき、常に宣言的か命令的なアプローチの選択肢があるかということです。そして、宣言的な解決策がある場合、それを常にコンピュータコードに翻訳できるかということです。この質問への答えは決して明白ではなく、見つけることができれば、おそらく宇宙の理解を革命的に変えるでしょう。

説明しましょう。物理学には

同様の双対性があり、それは何か深い台となる原則を指し示しているか、または私たちの心の働きについて何かを語っています。Richard Feynman は、彼の量子電磁力学に関する独自の研究でこの双対性に触発されたと述べています。



物理学のほとんどの法則を表現するには二つの形式があります。一つは局所的、または微小な考慮を使います。システムの状態を小さな近傍で見て、次の瞬間にどのように進化するかを予測します。これは通常、一定期間積分する必要のある微分方程式を使用して表現されます。

このアプローチが命令的思考に似ていることに注意してください：最終解を、それぞれが前のものの結果に依存する一連の小さなステップに従って到達します。実際、物理システムのコンピュータシミュレーションは、微分方程式を差分方程式に変換し、反復することに

よって定期的に実装されます。これは、アステロイドゲームで宇宙船がアニメーションされる方法です。各時間ステップで、宇宙船の位置は、速度を時間デルタで乗算して計算される小さな増分を加えて変更されます。速度は、力を質量で割った加速度に比例する小さな増分によって変更されます。

これらは Newton の運動法則に対応する微分方程式の直接のエンコーディングです:

$$F = m \frac{dv}{dt}$$
$$v = \frac{dx}{dt}$$

Maxwell 方程式を使用した電磁場の伝播や、格子 QCD (量子色力学) を使用した陽子内部のクォークとグルーオンの振る舞いなど、より複雑な問題にも同様の方法が適用されるかもしれません。

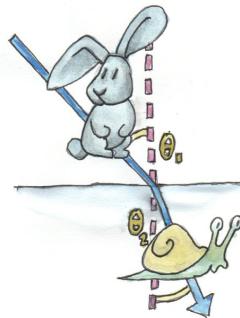
この局所的思考と、デジタルコンピュータの使用によって促される空間と時間の離散化は、Stephen Wolfram による宇宙の複雑さをセルラーオートマトンのシステムに還元しようとする英雄的な試みでその極限表現を見つけました。

もう一つのアプローチは大局的思考です。私たちはシステムの初期状態と最終状態を見て、ある機能を最小化することによってそれらを接続する軌道を計算します。最も単純な例は Fermat の最小時間の原理です。それは、光線が飛行時間を最小限にする経路を伝播すると述べています。特に、反射または屈折物体がない場合、点 A から点 B への光線は最短経路を取ります。これは直線です。しかし、光は密な(透

明な) 物質、例えば水やガラスでは遅く伝播します。ですから、始点を空気中に、終点を水中に置いた場合、空气中でより長く旅行し、水を通してショートカットを取る方が光にとって有利です。最小時間の経路は、光線が空気と水の境界で屈折し、結果として Snell の屈折法則が生じます:

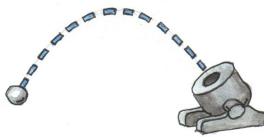
$$\frac{\sin(\theta_1)}{\sin(\theta_2)} = \frac{v_1}{v_2}$$

ここで v_1 は空気中の光速、 v_2 は水中の光速です。

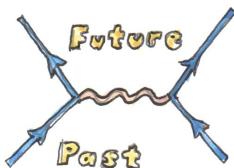


古典力学の全ては最小作用の原理から導出することができます。運動エネルギーと位置エネルギーの差 (注意: 和ではなく差です – 和は全エネルギーになります) である Lagrangian を積分することによって任意の軌道の作用を計算することができます。目標地点に向けて迫撃砲を発射する場合、弾丸はまず上に行き、重力による位置エネルギーが高いところでいくらかの時間を過ごし、作用に対して負の寄与を積み上げます。また、放物線の頂点で減速し、運動エネルギーを最小化し

ます。その後、低い位置エネルギーの領域を素早く通過するために加速します。



Feynman の最大の貢献は、最小作用の原理が量子力学に一般化できることを認識したことでした。そこでも、問題は初期状態と最終状態の観点から定式化されます。それらの状態間の Feynman 経路積分を使用して遷移の確率を計算します。



ポイントは、物理法則を記述する方法には不思議な双対性があり、それは局所的な絵で、物事は連続的に小さな増分で起こります。または、私たちは初期状態と最終状態を宣言し、その間のすべてはただ従うという大局的な絵を使うことができます。

大局的なアプローチはプログラミングでも使用することができます。例えば、レイトレーシングを実装する場合に使用します。目の位

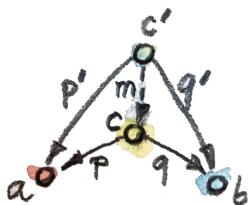
置と光源の位置を宣言し、光線がそれらを接続するために取り得る経路を計算します。私たちは各光線の飛行時間を明示的に最小化しませんが、Snell の法則と反射の幾何学と同じ効果で使用します。

局所的アプローチと大局的アプローチの最大の違いは、空間の扱いと、より重要なことに、時間の扱いにあります。局所的アプローチは、ここと今の即時の満足を受け入れますが、大局的なアプローチは長期的な静的な観点を取ります。未来が予め定められており、私たちは永遠の宇宙のいくつかの特性を分析しているかのようです。

このことは、ユーザーインターフェクションへの関数型リアクティブプログラミング (FRP) アプローチでよく例示されます。すべての可能なユーザーアクションに対して個別のハンドラーを書く代わりに、共有された変更可能な状態にアクセスするすべてのハンドラーが、外部イベントを無限リストとして扱い、それに一連の変換を適用します。概念的には、私たちの未来のすべてのアクションのリストがそこにあり、プログラムの入力データとして利用可能です。プログラムの観点からは、 π の桁のリスト、疑似ランダム数のリスト、またはコンピュータハードウェアを通じてくるマウスの位置のリストの間に違いはありません。それぞれの場合に n^{th} アイテムを取得したい場合、最初の $n - 1$ アイテムを通過する必要があります。時間的イベントに適用されるとき、私たちはこの特性を**因果性**と呼びます。

では、これが圏論とどのような関係があるのでしょうか？私は圏論が大局的なアプローチを奨励し、したがって宣言的プログラミングを支持すると主張します。まず第一に、微積分とは異なり、圏論には距

離や近傍、時間の組み込みの概念がありません。私たちが持っているのは抽象的な対象とそれらの間の抽象的な接続だけです。ある対象 A から B へ一連のステップを通じて到達できるなら、一つの跳躍でもそこに到達できます。さらに、圏論の主要なツールは普遍構成であり、それは大局的アプローチの極みです。例えば、圏論的積の定義においてそれが活用されているのを見ました。それはその特性によって指定されていました — 非常に宣言的なアプローチです。それは 2 つの射影を備えた対象であり、他のそのような対象の射影の因数分解を最適化する特性を持つ最良のそのような対象です。



これを Fermat の最小時間の原理や最小作用の原理と比較してください。

逆に、デカルト積の伝統的な定義と比較してみてください。それははるかに命令的です。一方の集合から一つの要素を選び、もう一方の集合から別の要素を選んで、ペアを作る方法を説明します。それはペアを作るためのレシピです。そして、ペアを分解するための別のレシピがあります。

ほとんどすべてのプログラミング言語、Haskell のような関数型言語を含むが、積型、余積型、関数型は普遍構成によって定義されるのではなく、組み込まれています。ただし、圏論的プログラミング言語を作る試みもあります(例えば、[Tatsuya Hagino の論文](#)^{*1}を参照)。

直接使用されるかどうかにかかわらず、圏論的定義は既存のプログラミング構造を正当化し、新しいものを生み出します。最も重要なことは、圏論がコンピュータプログラムについて宣言的レベルで推論するためのメタ言語を提供することです。また、問題の仕様についてコードに落とし込む前に推論することを促します。

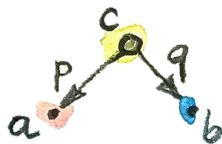
^{*1} <http://web.sfc.keio.ac.jp/~hagino/thesis.pdf>

12

極限と余極限

巻

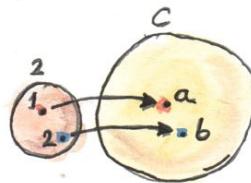
論では、すべてが何かしらと関連しており、多角的に見ることができます。例えば、**積**の普遍構成を取り上げてみましょう。関手や**自然変換**についてより深く理解すると、これを単純化し、場合によっては一般化できるのではないか？ 試してみましょう。



積の構成は、積を構成したい二つの対象 a と b を選択することから始まります。しかし、「対象を選択する」とはどういう意味でしょうか？

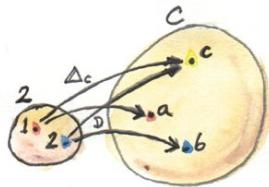
これをより圏論的な用語で言い換えることはできるでしょうか？二つの対象は非常に単純なパターンを形成します。このパターンを、非常に単純だけれども圏である $\mathbf{2}$ と呼ぶ圏に抽象化できます。これにはただ二つの対象 1 と 2 が含まれ、それ以外の射は二つの必須の恒等射だけです。これで、二つの対象を C で選択する行為を、圏 $\mathbf{2}$ から圏 C への関手 D を定義することと言えます。関手は対象を対象に写像しますから、その像はただの二つの対象です（一つになることもあります、それも問題ありません）。また、射も写像しますが、この場合は単に恒等射を恒等射に写像します。

このアプローチの素晴らしい点は、祖先の狩猟採集語彙から直接取られた「対象を選択する」といった不正確な表現を避けて、圏論の概念に基づいていることです。そして、偶然にも、 $\mathbf{2}$ より複雑な圏を使用してパターンを定義することを止めるものは何もないで、簡単に一般化できます。

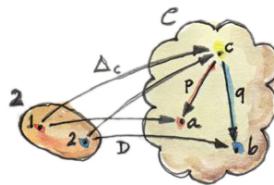


さて、積の定義の次のステップは、候補となる対象 c を選択することです。ここでも、单一の対象からの関手を用いて選択を言い換えることができます。そして実際、Kan 拡張を使用していれば、それが正し

い方法です。しかし、まだ Kan 拡張の準備ができていないので、別の技を使います: 同じ圏 2 から C への定数関手 Δ です。C で c を選択することは、 Δ_c で行えます。 Δ_c はすべての対象を c に、そしてすべての射を id_c に写像します。

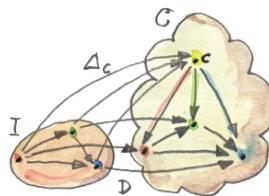


これで二つの関手、 Δ_c と D が 2 と C の間にできましたから、それらの間の自然変換について考えるのは自然です。2 には二つの対象しかないので、自然変換には二つのコンポーネントがあります。2 の対象 1 は Δ_c によって c に、そして D によって a に写像されます。したがって、 Δ_c と D の間の自然変換のコンポーネント 1 は、 c から a への射です。これを p と呼びましょう。同様に、二番目のコンポーネントは c から b への射 q です — D による 2 の対象 2 の像です。しかし、これらはまさに私たちが積の元の定義で使用した二つの射影のようです。したがって、対象と射影を選ぶ代わりに、関手と自然変換を選ぶだけよいのです。この単純な場合では、自然変換に対する自然性条件は、2 に恒等射以外の射がないため、自明に満たされます。

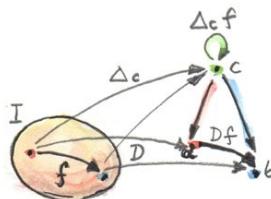


この構成を、 2 以外の圏、例えば非自明な射を含むものに一般化すると、 Δ_c と D の間の変換に自然性条件を課すことになります。そのような変換を錐と呼びます。なぜなら、 Δ の像は、自然変換のコンポーネントによって形成される錐／ピラミッドの頂点であり、 D の像が錐の底面を形成するからです。

一般に、錐を構成するには、パターンを定義する小さな、しばしば有限の圏 I から始めます。 I から C への関手 D を選び、それ(またはその像)を図式と呼びます。私たちの錐の頂点として何らかの c を C で選びます。それを使って、 I から C への定数関手 Δ_c を定義します。 Δ_c から D への自然変換が、私たちの錐です。有限の I に対しては、それは単に c を図式に接続するいくつかの射です: D による I の像です。



自然性は、この図のすべての三角形(ピラミッドの壁)が交差することを要求します。実際、 I の任意の射 f を取ります。関手 D はそれを C の射 Df に写像します。これは、いくつかの三角形の底面を形成する射です。定数関手 Δ_c は f を c 上の恒等射に写像します。 Δ は射の両端を一つの対象に押し潰し、自然性の四角形が交差する三角形になります。この三角形の二つの腕は自然変換のコンポーネントです。



これが一つの錐です。私たちが興味を持っているのは、**普遍錐**です—積の定義で私たちが選んだ普遍対象のように。

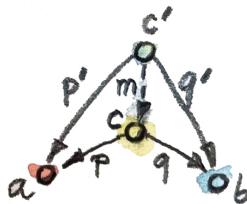
それにはいくつかの方法があります。例えば、与えられた関手 D に基づいて錐の圏を定義することができます。その圏の対象は錐です。しかし、 C のすべての対象 c が錐の頂点になるわけではありません。なぜなら、 Δ_c と D の間に自然変換が存在しないかもしれませんからです。

それを圏にするためには、錐の間の射も定義する必要があります。これらは頂点間の射によって完全に決定されます。しかし、どんな射でも良いわけではありません。私たちが積の構成で課した条件は、射影の共通因子である頂点間の射の条件でした。例えば:

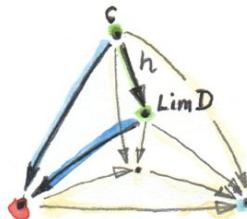
$$p' = p \circ m$$

$$q' = q \circ m$$

```
val p1 = p compose m  
val q1 = q compose m
```



この条件は、一般的な場合には、因子化射 (factorizing morphism) の一辺が三角形を形成する条件に翻訳されます。



二つの錐を接続する交差する三角形、因子化射 h (ここで、下の錐は普遍的なもので、その頂点は $\text{Lim } D$ です)

因子化射を錐の圏の射として取ります。これらの射が確かに合成され、恒等射が因子化射であることを確認するのは簡単です。したがって、錐は圏を形成します。

これで、普遍錐を錐の圏の終対象として定義することができます。終対象の定義は、他のどの対象からもその対象に一意の射があることを意味します。私たちの場合、それは他のどの錐の頂点からも普遍錐の頂点への一意の因子化射があることを意味します。この普遍錐を図式 D の極限、 $\text{Lim } D$ と呼びます（文献では、しばしば Lim 記号の下に I に向かって左向きの矢印を見ます）。しばしば、この錐の頂点を単に極限（または極限対象）と呼びます。

直感的には、極限は図式全体の特性を单一の対象に体現するということです。例えば、二つの対象の図式の極限は、二つの対象の積です。積（とその二つの射影と共に）は、両方の対象に関する情報を含んでいます。そして、普遍であるということは、余分なものが何も含まれていないことを意味します。

12.1 極限と自然同型

しかし、この極限の定義にはまだ満足できない点があります。確かに、これは実用的ですが、二つの錐を繋ぐ三角形の交換条件を何らかの自然性条件に置き換えることができれば、もっとエレガントになるでしょう。しかし、どうすればよいのでしょうか？

ここで我々は一つの錐ではなく、錐の全体的なコレクション(実際には、錐の圏)を扱っています。もし極限が存在する(そして—明らかにするが—それは保証されていない)、それらの錐の一つは普遍錐です。他のどの錐に対しても、その頂点を普遍錐の頂点である $\mathbf{Lim}D$ へ写像する特別な種類の一意の射があります(実際、「他の」という言葉を省略することができます。なぜなら、恒等射は普遍錐をそれ自身に写像し、それは自身を通して自明に因子化されるからです)。重要な部分を繰り返します: 任意の錐が与えられると、特別な種類の一意の射の写像があります。これは錐を特別な射への写像であり、それは一対一の写像です。

この特別な射はホム集合 $C(c, \mathbf{Lim}D)$ のメンバーです。このホム集合の他のメンバーは、二つの錐の写像を因子化しないという意味で不幸です。我々が望むのは、各 c に対して、特定の交換条件を満たす $C(c, \mathbf{Lim}D)$ の集合から一つの射を選ぶことができるようになります。これは自然変換を定義するようなものに聞こえますか? 確かにそうです!

しかし、この変換に関連する二つの関手は何でしょうか?

一つの関手は、 c を集合 $C(c, \mathbf{Lim}D)$ に写像するものです。これは C から \mathbf{Set} への関手です—対象を集合に写像します。実際、これは反変関手です。射の作用を定義するために、 c' から c への射 f を取ります:

$$f :: c' \rightarrow c$$

私たちの関手は c' を集合 $\mathbf{C}(c', \mathbf{Lim}D)$ に写像します。この関手の射 f に対する作用を定義するために(つまり、 f を持ち上げるために)、集合 $\mathbf{C}(c, \mathbf{Lim}D)$ と $\mathbf{C}(c', \mathbf{Lim}D)$ の間の対応する写像を定義する必要があります。そこで、集合 $\mathbf{C}(c, \mathbf{Lim}D)$ の一要素 u を選び、それを何らかの $\mathbf{C}(c', \mathbf{Lim}D)$ の要素に写像することができるかを見ます。ホム集合の要素は射ですから、我々は次のように持っています:

$$u :: c \rightarrow \mathbf{Lim}D$$

u を f で前合成すると、次のものが得られます:

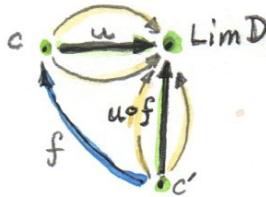
$$u \circ f :: c' \rightarrow \mathbf{Lim}D$$

そして、これは $\mathbf{C}(c', \mathbf{Lim}D)$ の要素です – したがって、確かに、我々は射の写像を見つけました:

```
contramap :: (c' -> c) -> (c -> LimD) -> (c' -> LimD)
contramap f u = u . f
```

```
def contramap[C, C1, D[_]](f: C1 => C, u: C => Lim[D]): (C1 => Lim[D]) =
  u compose f
```

c と c' の順序の反転に注意してください – これは反変関手の特徴です。



自然変換を定義するために、我々はもう一つの関手が必要で、それも C から Set への写像です。しかし今回は、錐の集合を考えます。錐はただの自然変換ですから、我々は自然変換 $\text{Nat}(\Delta_c, D)$ の集合を見ています。 c からこの特定の自然変換の集合への写像は、(反変) 関手です。それを示すにはどうすればよいでしょうか？再び、射に対するその作用を定義しましょう：

$$f :: c' \rightarrow c$$

f の持ち上げは、 I から C への二つの関手間の自然変換の写像でなければなりません：

$$\text{Nat}(\Delta_c, D) \rightarrow \text{Nat}(\Delta_{c'}, D)$$

自然変換をどのように写像しますか？ 各自然変換は射の選択です – そのコンポーネントは一つずつです – I の各要素に一つの射です。ある α ($\text{Nat}(\Delta_c, D)$ のメンバー) の I の対象 a におけるコンポーネントは射です：

$$\alpha_a :: \Delta_c a \rightarrow D a$$

または、定数関手 Δ の定義を使って、

$$\alpha_a :: c \rightarrow D a$$

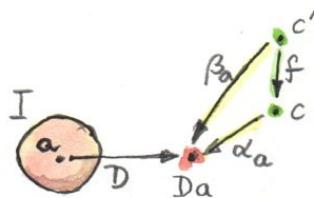
f と α が与えられた場合、 $Nat(\Delta_{c'}, D)$ のメンバーである β を構成する必要があります。その a におけるコンポーネントは次のような射でなければなりません：

$$\beta_a :: c' \rightarrow Da$$

我々は容易に後者 (β_a) を前者 (α_a) から前合成して f で得ることができます：

$$\beta_a = \alpha_a \circ f$$

これらのコンポーネントが実際に自然変換を加えることは比較的簡単に示せます。



射 f が与えられた我々は、二つの自然変換間の写像をコンポーネントごとに構成しました。この写像は関手の **contramap** を定義します：

$$c \rightarrow Nat(\Delta_c, D)$$

私が今示したのは、我々が **C** から **Set** への二つの(反変)関手を持って いるということです。私はどんな仮定もしていません – これらの関手は常に存在します。

ちなみに、これらの関手のうちの最初のものは圏論において重要な役割を果たし、我々が米田の補題について話すときに再び登場します。任意の圏 C から Set への反変関手には名前があります：それらは「前層」と呼ばれます。この一つは**表現可能前層**と呼ばれます。二番目の関手も前層です。

これで、我々は二つの関手を持っているので、それらの間の自然変換について話すことができます。それでは遠慮なく、ここが結論です： I から C への関手 D は、私が今定義した二つの関手の間に自然同型が存在する場合、そしてその場合に限り、極限 $\mathbf{Lim}D$ を持ります：

$$C(c, \mathbf{Lim}D) \simeq Nat(\Delta_c, D)$$

自然同型とは何かを思い出してください。それは、その全てのコンポーネントが同型、つまり可逆な射である自然変換です。

この声明の証明を通過するつもりはありません。その手順は、退屈であってもまっすぐです。自然変換を扱うときは、通常は射であるコンポーネントに焦点を当てます。この場合、両方の関手のターゲットが Set なので、自然同型のコンポーネントは関数です。これらは高次の関数です。なぜなら、それらはホム集合から自然変換の集合へ行くからです。再び、関数をその引数が何をするかで分析することができます：ここでは引数は射です — $C(c, \mathbf{Lim}D)$ のメンバーです — そして結果は自然変換です — $Nat(\Delta_c, D)$ のメンバー、または私たちが錐と呼んだものです。この自然変換は、その独自のコンポーネントを持って

おり、それらは射です。それで、それは射ですべての方法であり、それらを追跡することができれば、声明を証明することができます。

最も重要な結果は、この同型の自然性条件が、錐の写像の交換条件と正確に一致することです。

これからアトラクションのプレビューとして、集合 $\text{Nat}(\Delta_c, D)$ を関手圏のホム集合と見なすことができると言及しておきます。したがって、私たちの自然同型は二つのホム集合を関連付けており、これは随伴と呼ばれるさらに一般的な関係を指しています。

12.2 極限の例

我々は、単純な圏 $\mathbf{2}$ によって生成される図式の極限として、圏論的な積を見てきました。

さらに単純な極限の例があります: 終対象です。最初の衝動は、单一の対象から終対象に至ると考えるかもしれません、実際はそれよりも更にシンプルです: 終対象は空の圏によって生成される極限です。空の圏からの関手は対象を選択しませんので、錐は単に頂点だけになります。普遍錐は、他のどの頂点からも唯一の射が来る単独の頂点です。これは終対象の定義として認識されます。

次に興味深い極限はイコライザと呼ばれます。これは二つの要素と、それらの間に二つの平行な射がある二要素の圏によって生成される極限です(恒等射も常に含まれます)。この圏は \mathbf{C} の図式で、二つの対象 a と b 、そして二つの射を選択します:

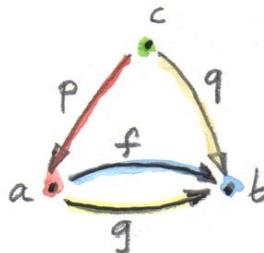
```
f :: a -> b  
g :: a -> b
```

```
val f : A => B  
val g : A => B
```

この図式の上に錐を構成するためには、頂点 c と二つの射を加える必要があります:

```
p :: c -> a  
q :: c -> b
```

```
val p : C => A  
val q : C => B
```



二つの三角形が交差する必要があります:

q = f . p

q = g . p

q == f compose p

q == g compose p

これは q がこれらの方程式の一つ、例えば $q = f . p$ によって一意に決まるることを教えてくれます。したがって、私たちは図からそれを省略することができます。ですから、私たちが残されるのは次の一つの条件です:

f . p = g . p

f compose p == g compose p

これについて考える方法は、我々が **Set** に注目を限定するならば、関数 p の像が a の部分集合を選択するということです。この部分集合に限定すると、関数 f と g は等しくなります。

例として、 a が x と y の座標によってパラメータ化される二次元平面を、 b を実数の線とし、次のように取ります:

f (x, y) = 2 * y + x

g (x, y) = y - x

```
def f(x, y) = 2 * y + x  
def g(x, y) = y - x
```

これら二つの関数のイコライザは、実数の集合 (頂点 c) と関数:

```
p t = (t, (-2) * t)
```

```
def p(t) = (t, (-2) * t)
```

です。 (pt) は二次元平面における直線を定義します。この線上で、二つの関数は等しくなります。

もちろん、他の集合 c' と関数 p' が存在して、等式:

```
f . p' = g . p'
```

```
f compose p1 == g compose p1
```

を満たすかもしれません、それらはすべて p を通じて一意に因子化されます。例えば、单集合 () を c' として、関数:

```
p'() = (0, 0)
```

```
def p1 : Unit => (Double, Double) = _ => (0, 0)
```

を取ることができます。これは $f(0, 0) = g(0, 0)$ なので良い錐です。しかし、それは h を通じて因子化されるため、普遍的ではありません:

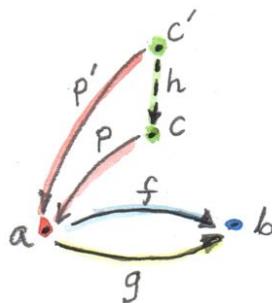
$p' = p . h$

val p1 = p compose h

以下のようになります:

$h \circ = 0$

def h(_,_) = 0



従って、イコライザは $f x = g x$ のような方程式を解くために使用することができます。しかし、それは物と射の観点で定義されているため、はるかに一般的です。

方程式を解くというさらに一般的なアイデアは、別の極限 - 引き戻しに体現されています。ここでも、私たちは等しいとしたい二つの射がありますが、今回はそれらの出発点が異なります。 $1 \rightarrow 2 \leftarrow 3$ の形

をした三要素の圏で始まります。この圏に対応する図式は三つの対象 a 、 b 、そして c 、そして二つの射から成ります：

$f :: a \rightarrow b$

$g :: c \rightarrow b$

`val f : A => B`

`val g : C => B`

この図式はしばしば余スパンと呼ばれます。

この図式の上に構成された錐は、頂点 d と三つの射から成ります：

$p :: d \rightarrow a$

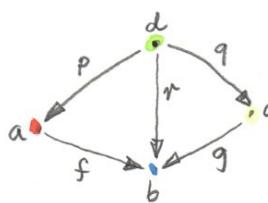
$q :: d \rightarrow c$

$r :: d \rightarrow b$

`val p : D => A`

`val q : D => C`

`val r : D => B`

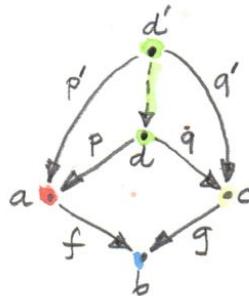


交換性の条件は r が他の射によって完全に決定されることを示しており、図から省略することができます。そこで、我々が残されるのは次の条件のみです：

$$g \cdot q = f \cdot p$$

$$g \text{ compose } q == f \text{ compose } p$$

引き戻しはこの形の普遍錐です。



もしあなたが集合に焦点を絞るならば、 d の対象を a と c の要素のペアで考えることができます。これらの要素のペアで、 f が第一要素に作用するのと同じ結果を、 g が第二要素に作用するとします。これが一般的には難しい場合は、 g が定数関数、たとえば $g_- = 1.23$ である特殊なケースを考えてみてください (b が実数の集合であると仮定します)。

この場合、あなたは実際に方程式：

$f(x) = 1.23$

$f(x) == 1.23$

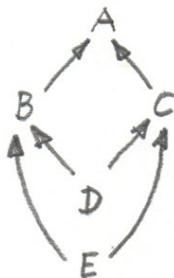
を解いています。

このケースでは、 c の選択は関係ありません (空集合でない限り)。ですから、 c として单集合を取ることができます。集合 a は例えば、三次元ベクトルの集合であり、そして f はベクトルの長さであるとします。すると、引き戻しは 1.23 の長さを持つベクトル v (方程式 $\sqrt{(x^2 + y^2 + z^2)} = 1.23$ の解) と、单集合のダミー要素 () のペア $(v, ())$ の集合になります。

しかし、引き戻しはプログラミングでもより一般的な用途を持っています。例えば、C++ のクラスを射がサブクラスからスーパークラスへの矢印となる図として考えます。継承を推移的な性質と見なし、もし C が B から、そして B が A から継承するならば、我々は C が A から継承すると言います (結局、 C へのポインターを A が要求される場所に渡すことができます)。また、 C が C 自身から継承するとも見なしますので、各クラスに対して恒等射があります。これにより、サブクラス化はサブタイピングと一致します。C++ はまた、複数継承をサポートしているので、 A から継承する B と C の二つのクラス、そして B と C から複数継承する第四のクラス D を持つダイヤモンド継承図を構成することができます。通常、 D は A の二つのコピーを得るでしょうが、

これはめったに望まれることではありません。しかし、仮想継承を使用して D 内の A のコピーを一つだけにすることができます。

D がこの図で引き戻しであることが意味するのは何でしょうか？ それは、B と C の両方から複数継承する任意のクラス E が、また D のサブクラスでもあるということを意味します。これは C++ では直接表現できません。サブタイピングは名目的であり、C++ コンパイラはこの種のクラス関係を推論しません – それは「ダックタイピング」が必要です。しかし、サブタイピング関係の外に出て、E から D へのキャストが安全かどうかを尋ねることはできます。このキャストは安全である可能性がありますが、D が B と C の裸の組み合わせであり、データを追加せず、メソッドをオーバーライドせずに、そして B と C の間にあるメソッドの名前の衝突がない場合に限ります。



また、型推論における引き戻しのより高度な使用もあります。二つの式の型を統合する必要がある場合がよくあります。例えば、コンパイラーが関数の型を推論したいとします：

```
twice f x = f (f x)
```

それはすべての変数と部分式に暫定的な型を割り当てます。特に、次のように割り当てます:

```
f      :: t0
x      :: t1
f x    :: t2
f (f x) :: t3
```

これから次のように推論します:

```
twice :: t0 -> t1 -> t3
```

関数適用の規則から一連の制約が生じます:

t0 = t1 -> t2 -- f が x に適用されるから

t0 = t2 -> t3 -- f が (f x) に適用されるから

これらの制約は、両方の式で未知の型に同じ型 (または型変数) を代入することにより同じ型を生成する一連の型 (または型変数) を見つけることで統合されなければなりません。そのような代入の一つは次のようになります:

```
t1 = t2 = t3 = Int
twice :: (Int -> Int) -> Int -> Int
```

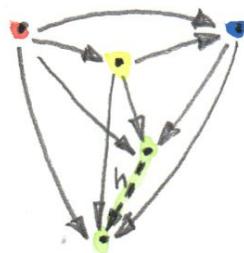
しかし、明らかに、これは最も一般的な代入ではありません。最も一般的な代入は引き戻しを使用して得られます。詳細に入ることはこの本の範囲を超えてますが、結果として次のようになることを自分で納得させることができます:

```
twice :: (t -> t) -> t -> t
```

ここで t は自由な型変数です。

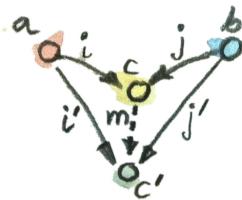
12.3 余極限

圏論の全ての構成は、逆圏においてその双対像を持ちます。錐の全ての射の方向を逆にすると、余錐が得られ、その普遍的なものが余極限と呼ばれます。因子化射も反転され、普遍余錐から他の任意の余錐へ流れます。



因子化射 h が二つの頂点を接続する余錐。

余極限の典型的な例は余積であり、積の定義に使用した 2 の圏によって生成された図式に対応します。



積と余積はそれぞれ異なる方法で一対の対象の本質を体現しています。

終対象が極限であったように、始対象は空の圏に基づく図式の余極限に対応する余極限です。

引き戻しの双対は押し出しと呼ばれます。それは $1 \leftarrow 2 \rightarrow 3$ の形の圏によって生成される図式、スパンと呼ばれる図式に基づいています。

12.4 連続性

私は以前、関手が圏の連続写像のアイデアに近いものであると述べました。それは、既存の接続(射)を決して破壊しないという意味でです。実際の連続関手 F の定義は、 C から C' への関手が極限を保つという要件を含んでいます。 C の任意の図式 D は、単純に二つの関手を合成することによって C' の図式 $F \circ D$ に写像されます。 F の連続性条

件は、図式 D が極限 $\mathbf{Lim}D$ を持つ場合、図式 $F \circ D$ も極限を持ち、それが $F(\mathbf{Lim}D)$ であることを要求します。



関手が射を射に写像し、合成を合成に写像するので、錐の像は常に錐です。交差する三角形は常に交差する三角形に写像されます(関手は合成を保存します)。因子化射も同様です: 因子化射の像も因子化射です。だから、全ての関手はほとんど連続です。何が間違っていく可能性があるかというと、一意性の条件です。 C' における因子化射は一意であるとは限りません。 C' には、 C には存在しなかった他の「より良い錐」もあるかもしれません。

ホム関手は連続関手の一例です。ホム関手、 $C(a, b)$ は、最初の変数において反変であり、第二の変数において共変です。言い換えれば、それは関手です:

$$C^{op} \times C \rightarrow \mathbf{Set}$$

第二引数が固定されると、ホム集合関手(表現可能前層になる)は C における余極限を \mathbf{Set} における極限に写像します。第一引数が固定されると、極限を極限に写像します。

Haskellにおいて、ホム関手は任意の二つの型を関数型に写像するものです。ですから、それは単にパラメータ化された関数型です。第

二パラメータを固定すると、例えば `String` にすると、反変関手が得られます:

```
newtype ToString a = ToString (a -> String)
instance Contravariant ToString where
    contramap f (ToString g) = ToString (g . f)

trait Contravariant[F[_]] {
    def contramap[A, B](fa: F[A])(f: B => A) : F[B]
}

class ToString[A](f: A => String) extends AnyVal

implicit val contravariant = new Contravariant[ToString] {
    def contramap[A, B](fa: ToString[A])(f: B => A): ToString[B] =
        ToString(fa.f compose f)
}
```

連続性は、例えば `ToString` が余積、例えば `Either b c` を適用されると極限を生み出すことを意味します；このケースでは二つの関数型の積です:

```
ToString (Either b c) ~ (b -> String, c -> String)
```

```
ToString[Either[B, C]] ~ (B => String, C => String)
```

実際、任意の `Either b c` の関数は、二つのケースをサービスする一対の関数によるケース文として実装されます。

同様に、

ホム集合の第一引数を固定すると、お馴染みの Reader 関手が得られます。その連續性は、例えば、積を返す任意の関数が関数の積に相当することを意味します。特に:

$$r \rightarrow (a, b) \sim (r \rightarrow a, r \rightarrow b)$$

$$R \Rightarrow (A, B) \sim (R \Rightarrow A, R \Rightarrow B)$$

私が知っているあなたが考えていることは、これらのこと理解するのに圏論は必要ないということです。そして、あなたは正しいです！それでも、これらの結果がビットとバイト、プロセッサーーアーキテクチャ、コンパイラ技術、さらにはラムダ計算に一切頼ることなく、第一原理から導出されることに私は驚嘆します。

「極限」と「連續性」の名前がどこから来ているのか気になる場合、それらは微積分からの対応する概念の一般化です。微積分における極限と連續性は開集合に関して定義されます。開集合はトポロジーを定義し、トポロジーは圏(半順序集合)を形成します。

12.5 チャレンジ

1. C++ のクラスの圏において押し出しをどのように説明しますか？
2. 恒等関手 $\text{Id} :: \mathbf{C} \rightarrow \mathbf{C}$ の極限が始対象であることを示してください。
3. 与えられた集合の部分集合は圏を形成します。この圏における射は、一つがもう一つの部分集合である二つの集合を結ぶ矢印として定義されます。このような圏における二つの集合の引き戻しは何ですか？ 押し出しこれ何ですか？ 始対象と終対象は何かですか？
4. 余イコライザが何であるか推測できますか？
5. 終対象が存在する圏において、終対象に向けての引き戻しが積であることを示してください。
6. 同様に、始対象からの押し出しが(存在する場合)余積であることを示してください。

13

自由モノイド

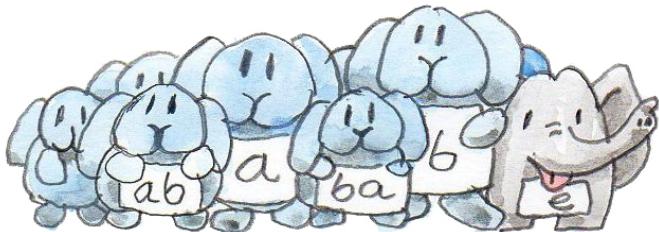
巻論およびプログラミングにおいて、モノイドは重要な概念です。圏は厳密な型付け言語に対応し、モノイドは非型付け言語に対応します。それはモノイドでは任意の二つの矢印を合成できるように、非型付け言語では任意の二つの関数を合成できるからです（もちろん、プログラムを実行した時に実行時エラーになるかもしれません）。

单一の対象を持つ圏としてモノイドを記述できることがわかりました。ここでは、射の合成則に全てのロジックがエンコードされています。この圏論的モデルは、集合を「掛ける」伝統的な集合論的定義のモノイドと完全に等価です。二つの集合の要素を掛け合わせて第三の要素を得るこのプロセスは、さらに分解して、最初に要素のペアを形

成し、次にこのペアを既存の要素と同一視することによって「積」が得られます。

積の第二部分 – ペアを既存の要素と同一視すること – を省略したらどうなるでしょうか？ たとえば、任意の集合から始めて、要素の全可能なペアを形成し、それらを新しい要素と呼ぶことができます。次に、これらの新しい要素を全可能な要素とペアにし、という具合に続けます。これは連鎖反応です – 新しい要素を永遠に追加し続けます。結果として無限集合が得られますが、それはほぼモノイドです。しかしモノイドには単位要素と結合則も必要です。問題ないです、特別な単位要素を追加し、ペアのいくつかを同一視するだけで、単位則と結合則をサポートすることができます。

シンプルな例でこの仕組みを見てみましょう。二つの要素 $\{a, b\}$ の集合から始めます。これらを自由モノイドの生成器と呼びます。まず、単位として機能する特別な要素 e を追加します。次に、要素のペアを全て追加して「積」と呼びます。 a と b の積はペア (a, b) になります。 b と a の積はペア (b, a) 、 a と a の積は (a, a) 、 b と b の積は (b, b) になります。 e とペアを形成することができますが、 (a, e) 、 (e, b) などは a 、 b などと同一視します。したがって、このラウンドでは (a, a) 、 (a, b) 、 (b, a) 、 (b, b) のみを追加し、集合 $\{e, a, b, (a, a), (a, b), (b, a), (b, b)\}$ を得ます。



次のラウンドでは、 $(a, (a, b))$ 、 $((a, b), a)$ などの要素を追加し続けます。この時点で結合則が成り立つことを確認する必要がありますから、 $(a, (b, a))$ を $((a, b), a)$ などと同一視します。言い換えると、内部の括弧は必要ありません。

このプロセスの最終結果を想像できるでしょう： a と b の全可能なリストを作成します。実際、 e を空リストとして表現すれば、私たちの「掛け算」はリストの連結に他なりません。

このような構成、つまり要素の全可能な組み合わせを生成し、規則をサポートするために必要最小限の同一視を行うことは、自由構成と呼ばれます。私たちが行ったことは、生成器の集合 $\{a, b\}$ から**自由モノイド**を構成することでした。

13.1 Haskell における自由モノイド

Haskell における二要素の集合は `Bool` 型に相当し、この集合によって生成される自由モノイドは `[Bool]` 型 (`Bool` のリスト) に相当します。(私は無限リストの問題を意図的に無視しています。)

Haskell におけるモノイドは型クラスによって定義されます:

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m

trait Monoid[M] {
    def mempty: M
    def mappend(m1: M, m2: M): M
}
```

これは、すべての `Monoid` には `mempty` と呼ばれる中立要素と、`mappend` と呼ばれる二項関数(掛け算)が必要であることを意味します。単位則と結合則は Haskell では表現できず、モノイドをインスタンス化するたびにプログラマによって検証されなければなりません。

任意の型のリストがモノイドを形成するという事実は、このインスタンス定義によって記述されています:

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

```
object Monoid {  
    implicit def listMonoid[A]: Monoid[List[A]] = new Monoid[List[A]] {  
        def mempty: List[A] = List()  
        def mappend(m1: List[A], m2: List[A]): List[A] = m1 ++ m2  
    }  
}
```

これは、空リスト [] が単位要素であり、リストの連結 (++) が二項演算であることを示しています。

見てきたように、型 **a** のリストは、生成器としての集合 **a** を持つ自由モノイドに対応しています。自然数の集合とその積は自由モノイドではありません。なぜなら、多くの積が同一視されるからです。例えば:

```
2 * 3 = 6  
[2] ++ [3] = [2, 3] // not the same as [6]  
  
2 * 3 == 6  
List(2) ++ List(3) == List(2, 3)
```

これは簡単ですが、問題は、圏論において、私たちが対象の内部を見ることが許されない場合に、この自由構成を行うことができるかということです。私たちは普遍構成を使用します。

二つ目の興味深い問題は、任意のモノイドが、規則が要求する最小限の要素数以上を同一視することによって、何らかの自由モノイドから得られるかどうかです。普遍構成から直接導かれることを示します。

13.2 自由モノイドの普遍構成

以前の普遍構成の経験を振り返ると、それが何かを構成することよりも、与えられたパターンに最も適合する対象を選択することについてであることに気付くかもしれません。ですから、自由モノイドを「構成」するために普遍構成を使用する場合、私たちは選択するために一連のモノイドを考慮しなければなりません。私たちは選択するために、モノイドの全圏が必要です。しかし、モノイドは圏を形成しますか？

まず、単位と掛け算によって定義される追加構造を備えた集合としてモノイドを見てみましょう。モノイダル構造を保存する関数を射として選びます。このような構造保存関数は**準同型**と呼ばれます。モノイドの準同型は、二つの要素の積を、二つの要素のマッピングの積にマップしなければなりません:

$$h(a * b) = h a * h b$$

$$h(a * b) == h(a) * h(b)$$

そして、単位を単位にマップしなければなりません。

たとえば、整数のリストから整数への準同型を考えてみましょう。もし `[2]` を 2 に、`[3]` を 3 にマップするなら、`[2, 3]` を 6 にマップしなければなりません。なぜなら、連結

```
[2] ++ [3] = [2, 3]
```

```
List(2) ++ List(3) == List(2, 3)
```

は掛け算になります

```
2 * 3 = 6
```

```
2 * 3 == 6
```

では、個々のモノイドの内部構造を忘れて、それらを対応する射とともに対象としてのみ見ます。**Mon** のモノイドの圏を得ます。

まあ、内部構造を忘れる前に、重要な特性に気づいておきましょう。**Mon** のすべての対象は、自明な方法で集合にマッピングすることができます。それは単にその要素の集合です。この集合は**台となる集合**と呼ばれます。実際、**Mon** の対象だけでなく、**Mon** の射（準同型）も **Set** にマップすることができます。これは種類のトリビアルなように思えるかもしれません、すぐに役立ちます。この対象と射のマッピングは実際には関手です。この関手はモノイダル構造を「忘れる」ため、一度ただの集合の中に入れば、単位要素を区別したり、掛け算を

気にしたりする必要はありません。それで、これは忘却関手と呼ばれます。忘却関手は圏論において定期的に現れます。

私たちは今、**Mon** を二つの異なる視点から見ることができます。対象と射のある他の圏と同じにそれを扱うことができます。この視点では、モノイドの内部構造を見ることはできません。**Mon** の特定の対象について言えるのは、それが自身および他の対象に対して射を通じて接続されていることだけです。射の「掛け算」表 – 合成則 – は他の視点、つまり集合としてのモノイドから派生します。圏論に移行してもこの視点を完全には失っていません – 忘却関手を通じてそれにアクセスできます。

普遍構成を適用するには、モノイドの圏を通じて探索し、自由モノイドに最適な候補を選ぶための特別な特性を定義する必要があります。しかし、自由モノイドはその生成器によって定義されます。異なる生成器の選択は異なる自由モノイドを生み出します (**Bool** のリストは **Int** のリストとは異なります)。私たちの構成は生成器の集合から始める必要があります。だから、再び集合に戻ります！

ここで忘却関手が役立ちます。それを使って私たちのモノイドを X 線写真で見ることができます。それらのプロップの X 線画像内で生成器を識別することができます。こうして機能します:

私たちは、**Set** 内の生成器の集合 x から始めます。

私たちが一致させようとするパターンは、**Mon** の対象であるモノイド m と **Set** 内の関数 p で構成されます:

`p :: x -> U m`

`val p: X => U[M]`

ここで、 U は **Mon** から **Set** への忘却関手です。これは **Mon** と **Set** の半分が混在した奇妙な異種パターンです。

この考えは、関数 p が m の X 線画像内の生成器を識別するというものです。関数が集合内の点を識別するのが下手でも問題ありません（それらは点を崩壊させるかもしれません）。普遍構成によって、このパターンの最良の代表者を選ぶことがすべて整理されます。



候補者の中からランク付けするために、別の候補者を考えてみましょう：モノイド n と、その X 線画像内で生成器を識別する関数：

`q :: x -> U n`

`val q: X => U[N]`

m が n よりも優れていると言えるのは、モノイドの射（構造保存準同型）が存在し：

h :: m -> n

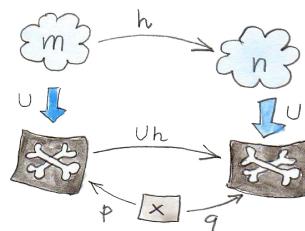
val h: M => N

その U 下での像 (U は関手なので、射を関数にマップします) が p を介して分解される場合です:

q = U h . p

val q = uh compose p

p が m 内の生成器を選び、 q が n 内の「同じ」生成器を選ぶと考えると、 h はこれらの生成器を二つのモノイド間でマッピングしていると考えることができます。 h は、定義上、モノイダル構造を保存します。つまり、一つのモノイド内の二つの生成器の積は、二番目のモノイド内の対応する二つの生成器の積にマッピングされることを意味します。



このランキングは、自由モノイドである最良の候補を見つけるために使用されます。定義は次のようにになります：

m (および関数 p) が生成器 x を持つ**自由モノイド**であると言えるのは、 m から他の任意のモノイド n (および関数 q) への一意の射 h が存在し、上記の分解特性を満たす場合のみです。

ちなみにこれは私たちの二番目の質問にも答えます。 Uh は、 Um の多くの要素を Un の単一の要素に崩壊させる力を持っています。この崩壊は、自由モノイドのいくつかの要素を同一視することに対応します。したがって、生成器 x を持つ任意のモノイドは、自由モノイドが x に基づいている場合、いくつかの要素を同一視することによって得られます。自

由モノイドは、最小限の同一視のみが行われたものです。

自由モノイドについては、随伴について話すときに戻ってきます。

13.3 チャレンジ

- あなたは (私が最初にしたように) モノイドの準同型が単位を保存する要件が冗長だと思うかもしれません。なぜなら、すべての a に対して

$$h a * h e = h (a * e) = h a$$

ですから、 he は右単位(そして類推により左単位)のように振る舞います。問題は、すべての a に対する ha が、ターゲットモノイドの部分モノイドをカバーするだけかもしれないということです。 h の像の外側に「真の」単位が存在するかもしれません。乗法を保存するモノイド間の同型が自動的に単位を保存することを示してください。

2. 整数のリストと連結に対するモノイド準同型を整数とその積に対するモノイド準同型として考えてみてください。空リスト [] の像は何ですか？すべての単一リストがそれらが含む整数にマップされると仮定します。つまり、[3] は 3 にマップされます。では、[1, 2, 3, 4] の像は何でしょうか？何個の異なるリストが整数 12 にマップされますか？他にも両方のモノイド間の準同型が存在しますか？
3. 単集合によって生成される自由モノイドは何ですか？それが何と同型であるかわかりますか？

14

表現可能関手

そろそろ集合について少し話しましょう。数学者は集合理論と愛憎関係にあります。それは数学のアセンブリ言語のようなものですー少なくとも昔はそうでした。圏論は、ある程度、集合理論から距離を置こうとします。例えば、全ての集合の集合は存在しないというのが知られていますが、全ての集合の圏、**Set** は存在します。だからそれは良いことです。一方で、圏の任意の二つの対象間の射は集合を形成すると仮定しています。我々はそれをホム集合と呼びました。公平を期すと、射が集合を形成しない圏論の分岐もあります。その代わりに、それらは別の圏の対象です。ホム集合ではなくホム対象を使用

するそれらの圏は、豊穣な圏と呼ばれます。しかし、これから我々が扱うのは、良い古典的なホム集合を持つ圏です。

集合は、圏論的対象の外で得られる、ほとんど特徴のないプロズに最も近いものです。集合には要素がありますが、これらの要素についてはあまり語ることができません。有限集合があれば、要素を数えることができます。無限集合の要素を、基数を使って何となく数えることができます。例えば、自然数の集合は実数の集合よりも小さいですが、どちらも無限です。しかし、驚くべきことに、有理数の集合は自然数の集合と同じ大きさです。

それ以外に、集合に関する全ての情報は、それらの間の関数 – 特に同型と呼ばれる可逆なもの – にエンコードすることができます。全ての意図と目的において、同型な集合は同一です。基礎數学者の怒りを呼ぶ前に、等価性と同型の違いは、最新の数学の分岐、ホモトピー型理論 (HoTT) の主要な関心事の一つです。HoTT は計算からインスピレーションを得た純粋な数学理論であり、その主要な提唱者の一人である Vladimir Voevodsky は、Coq 定理証明器を研究中に主要な啓示を得ました。数学とプログラミングの相互作用は双方向です。

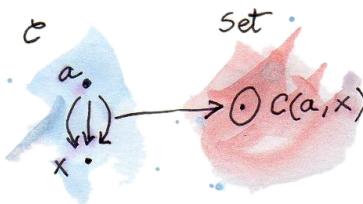
集合に関する重要な教訓は、異なる要素の集合を比較することが許されるということです。たとえば、ある自然変換の集合が、ある射の集合と同型であると言えます、なぜなら集合はただの集合だからです。この場合の同型とは、一方の集合からの自然変換ごとに他方の集合から一意の射があり、その逆も同様であることを意味します。それらは互いにペアにすることができます。異なる圏の対象であれば、リンゴ

とオレンジを比較することはできませんが、リンゴの集合とオレンジの集合を比較することはできます。しばしば、圏論的な問題を集合理論的な問題に変換することで、必要な洞察を得たり、貴重な定理を証明することができます。

14.1 ホム関手

すべての圏は、**Set** への標準的な写像の族を備えています。これらの写像は実際には関手であり、したがって圏の構造を保持します。そのような写像を一つ構成してみましょう。

C の一つの対象 a を固定し、 C 内の別の対象 x を選びます。ホム集合 $C(a, x)$ は集合であり、**Set** の対象です。 x を変えると、 a を固定したまま、 $C(a, x)$ も **Set** 内で変わります。したがって、 x から **Set** への写像があります。



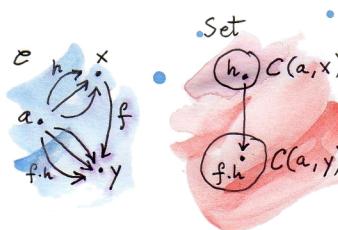
ホム集合を第二引数の写像として考慮したい場合、記法 $C(a, -)$ を使用し、ダッシュが引数のプレースホルダーとして機能します。

この対象の写像は容易に射の写像に拡張されます。 \mathbf{C} 内の任意の射 f を、任意の対象 x と y の間で取りましょう。対象 x は集合 $\mathbf{C}(a, x)$ に写され、対象 y は私たちが定義した写像の下で $\mathbf{C}(a, y)$ に写されます。この写像が関手であるためには、 f は二つの集合間の関数に写されなければなりません: $\mathbf{C}(a, x) \rightarrow \mathbf{C}(a, y)$

この関数を点ごとに定義しましょう、つまり各引数に対して別々に。引数として $\mathbf{C}(a, x)$ の任意の要素を選びましょう — それを h と呼びます。射は、端と端が一致する場合に合成可能です。偶然にも f の源が h の目標と一致するので、その合成:

$$f \circ h :: a \rightarrow y$$

は a から y への射です。したがって、それは $\mathbf{C}(a, y)$ のメンバーです。



私たちはちょうど $\mathbf{C}(a, x)$ から $\mathbf{C}(a, y)$ への関数を見つけました、これが f の像として機能することができます。混乱の恐れがなければ、この持ち上げられた関数を $\mathbf{C}(a, f)$ と書き、射 h に対するその作用を次のように書きます:

$$\mathbf{C}(a, f)h = f \circ h$$

この構造が任意の圏で機能するので、Haskell の型の圏にも機能します。Haskell では、ホム関手は **Reader** 関手としてよりよく知られています:

```
type Reader a x = a -> x

type Reader[A, X] = A => X

instance Functor (Reader a) where
    fmap f h = f . h

implicit def readerFunctor[A] = new Functor[Reader[A, ?]] {
    def fmap[X, B](f: X => B)(h: Reader[A, X]): Reader[A, B] =
        f compose h
}
```

それでは、ホム集合の源を固定するのではなく、目標を固定した場合に何が起こるかを考えてみましょう。つまり、写像 $C(-, a)$ も関手であるかという問いです。それはそうですが、共変であるのではなく反変です。それは、射の端と端の同じ種類の一致が、 $C(a, -)$ の場合とは逆に、 f による後方合成をもたらすからです。

私たちはすでに Haskell でこの反変関手を見てきました。それは **Op** と呼ばれていました:

```
type Op a x = x -> a

type Op[A, X] = X => A

instance Contravariant (Op a) where
    contramap f h = h . f

implicit def opContravariant[A] = new Contravariant[Op[A, ?]] {
    def contramap[X, B](f: B => X)(h: Op[A, X]): Op[A, B] =
        h compose f
}
```

最終的に、両方の対象を変化させると、第一引数で反変で、第二引数で共変であるプロ関手 $C(-,=)$ が得られます (二つの引数が独立して変化することを強調するために、第二プレースホルダーとしてダブルダッシュを使用します)。私たちは以前、関手性について話をしたときにこのプロ関手を見ました:

```
instance Profunctor (->) where
    dimap ab cd bc = cd . bc . ab
    lmap = flip (.)
    rmap = (.)

implicit def arrowProfunctor = new Profunctor[Function1] {
    def dimap[A, B, C, D](ab: A => B)(cd: C => D)(bc: B => C): A => D =
```

```

cd compose bc compose ab

def lmap[A, B, C](f: C => A)(ab: A => B): C => B =
  f andThen ab

def rmap[A, B, C](f: B => C)(ab: A => B): A => C =
  f compose ab
}

```

重要な教訓は、この観察が任意の圏で成り立つということです：対象をホム集合に写す写像は関手的です。反変性が逆圏からの写像に相当するので、私たちはこの事実を簡潔に述べることができます：

$$C(-, =) :: \mathbf{C}^{op} \times \mathbf{C} \rightarrow \mathbf{Set}$$

14.2 表現可能関手

\mathbf{C} の任意の対象 a を選ぶと、 \mathbf{C} から \mathbf{Set} への関手が得られます。このような \mathbf{Set} への構造保存写像はしばしば**表現**と呼ばれます。 \mathbf{C} の対象と射を \mathbf{Set} の集合と関数として表現しています。

関手 $C(a, -)$ 自体は時々表現可能と呼ばれます。より一般的に、ホム関手に自然に同型であるような任意の関手 F は、**表現可能**と呼ばれます。そのような関手は必然的に \mathbf{Set} 値でなければならず、 $C(a, -)$ がそうであるからです。

私は以前、私たちは通常同型な集合を同一と考えると言いました。より一般的には、圏内の同型な**対象**を同一と考えます。それは、対象

が射を通じて他の対象 (およびそれ自身) との関係以外に構造を持たないからです。

例えば、私たちは以前、集合で初めてモデル化されたモノイドの圏、**Mon**について話しました。しかし、我々はそれらの集合のモノイダル構造を保存する関数のみを射として選びました。したがって、**Mon** 内の二つの対象が同型である場合、つまりそれらの間に可逆な射がある場合、それらは正確に同じ構造を持っています。我々がそれらの基礎となる集合と関数を覗き見れば、一つのモノイドの単位素が別のモノイドの単位要素に写され、二つの要素の積がその写像の積に写されるのを見るでしょう。

関手に対する同じ推論を適用することができます。二つの圏間の関手は、自然変換が射として機能する圏で形成されます。したがって、二つの関手が同型であり、それらが同一であると考えることができます。それらの間に可逆な自然変換がある場合。

表現可能関手の定義をこの視点から分析しましょう。 F が表現可能であるためには、次が必要です: C の中に対象 a が存在し、 $C(a, -)$ から F への自然変換 α が一つ、逆方向にはもう一つの自然変換 β があり、そしてそれらの合成が恒等自然変換であること。

α の x におけるコンポーネントを見てみましょう。それは **Set** 内の関数です:

$$\alpha_x :: C(a, x) \rightarrow Fx$$

この変換の自然性条件は、 x から y への任意の射 f に対して、次の図式が可換であることを告げています：

$$Ff \circ \alpha_x = \alpha_y \circ C(a, f)$$

Haskellでは、自然変換を多相的関数で置き換えます：

```
alpha :: forall x. (a -> x) -> F x

// In order to make a universal transformation,
// another type needs to be introduced.
// To read more about FunctionK (~>):
// typelevel.org/cats/datatypes/functionk.html
trait ~>[F[_], G[_]] {
  def apply[B](fa: F[B]): G[B]
}

// Kind Projector plugin provides
// a more concise syntax for type lambdas:
def alpha[A]: (A => ?) ~> F
```

オプショナルの `forall` 量化子を使います。自然性条件

```
fmap f . alpha = alpha . fmap f
```

```
(fmap(f) _ compose alpha.apply) == (alpha.apply _ compose fmap(f))
```

は、パラメトリシティにより自動的に満たされます(私が以前言及した無料の定理の一つです)、左側の `fmap` は関手 F によって定義され、右側は Reader 関手によって定義されることを理解しています。Reader のための `fmap` はただの関数前置合成なので、より明確にすることができます。 $C(a, x)$ の要素である h に作用すると、自然性条件は次のように単純化されます:

```
fmap f (alpha h) = alpha (f . h)
```

```
fmap(f)(alpha(h)) == alpha(f compose h)
```

逆方向の変換、`beta` は反対の方向に進みます:

```
beta :: forall x. F x -> (a -> x)
```

```
def beta[A]: F ~> (A => ?)
```

それは自然性条件を尊重しなければならず、`alpha` の逆でなければなりません:

```
alpha . beta = id = beta . alpha
```

私たちは後で、 Fa が空でない限り、 $C(a, -)$ から任意の Set 値関手への自然変換が常に存在することを見るでしょう（米田の補題）が、それが必ずしも可逆ではないことも見るでしょう。

Haskell でのリスト関手と `Int` を `a` とする例を挙げましょう。ここでは仕事をする自然変換を次のように任意に選びます：

```
alpha :: forall x. (Int -> x) -> [x]
alpha h = map h [12]
```

```
def alpha: (Int => ?) ~> List = new ~>[Int => ?, List] {
  def apply[A](fa: Int => A) =
    List(12).map(fa)
}
```

私は任意に数 12 を選び、それで一つの要素のリストを作りました。その後、関数 `h` をこのリスト上で `fmap` することができ、`h` によって返される型のリストを得ることができます。（実際には、整数のリストと同じ数だけこのような変換が存在します。）

自然性条件は `map` の合成可能性 (`fmap` のリストバージョン) と同等です：

```
map f (map h [12]) = map (f . h) [12]
```

```
fmap(f)(fmap(h)(List(12))) == fmap(f compose h)(List(12))
```

```
// Or using scala stdlib:  
List(12).map(h).map(f) == List(12).map(f compose h)
```

しかし、逆変換を見つけようとした場合、任意の型 x のリストから x を返す関数へ行かなければなりません：

```
beta :: forall x. [x] -> (Int -> x)  
  
def beta: List ~> (Int => ?)
```

例えば、**head** を使ってリストから x を取り出すことを考えるかもしれません、空のリストではうまくいきません。このため、**a** の型として (Int の代わりに) 機能する選択肢はありません。従ってリスト関手は表現可能ではありません。

Haskell の (自己) 関手がコンテナのようなものだと話したときのことを覚えていますか？ 同じ意味で、表現可能関手は関数呼び出しのメモ化結果を格納するコンテナとして考えることができます。Haskell ではホム集合のメンバーは単なる関数です。表現する対象、 $C(a, -)$ の型 a は、関数の表計算値にアクセスするためのキー型と考えられます。私たちが **alpha** と呼んだ変換は **tabulate** と呼ばれ、その逆は **index** と呼ばれます。ここに (少し単純化された) **Representable** クラスの定義があります：

```
class Representable f where  
  type Rep f :: *
```

```
tabulate :: (Rep f -> x) -> f x
index     :: f x -> Rep f -> x

trait Representable[F[_]] {
  type Rep
  def tabulate[X](f: Rep => X): F[X]
  def index[X]: F[X] => Rep => X
}
```

表している型、つまり我々の a である **Rep f** は、

Representable の定義の一部であることに注意してください。星印は単に **Rep f** が型であることを意味します(型コンストラクタや他のもっと奇妙なものとは対照的に)。

空でない無限リスト、またはストリームは表現可能です。

```
data Stream x = Cons x (Stream x)

case class Stream[X](
  h: () => X,
  t: () => Stream[X]
)
```

それらは引数として **Integer** を取る関数のメモ化値として考えることができます。(厳密に言えば、非負の自然数を使うべきですが、コードを複雑にしたくありませんでした。)

そのような関数を `tabulate` するためには、値の無限ストリームを作成します。もちろん、これは Haskell が遅延評価であるために可能です。値は要求に応じて評価されます。`index` を使ってメモ化された値にアクセスします:

```
instance Representable Stream where
    type Rep Stream = Integer
    tabulate f = Cons (f 0) (tabulate (f . (+1)))
    index (Cons b bs) n = if n == 0 then b else index bs (n - 1)

implicit val streamRepresentable = new Representable[Stream] {
    type Rep = Int

    def tabulate[X](f: Rep => X): Stream[X] =
        Stream[X](() => f(0), () => tabulate(f compose (_ + 1)))

    def index[X]: Stream[X] => Rep => X = {
        case Stream(b, bs) =>
            n =>
                if (n == 0) b()
                else index(bs())(n - 1)
    }
}
```

興味深いことに、任意の戻り型の全ての族の関数に対して单一のメモ化スキームを実装することができます。

反変関手に対する表現可能性も同様に定義されますが、 $\mathbf{C}(-, a)$ の第二引数を固定します。または、 \mathbf{C}^{op} から \mathbf{Set} への関手として考えることができます、なぜなら $\mathbf{C}^{op}(a, -)$ は $\mathbf{C}(-, a)$ と同じだからです。

表現可能性には面白い捩れがあります。ホム集合は、デカルト閉圏内で指標対象として内部的に扱うことができるのを覚えていてください。ホム集合 $\mathbf{C}(a, x)$ は x^a に相当し、表現可能関手 F に対して、私たちには $-^a = F$ と書くことができます。

両側に対数を取ってみましょう、単にキックのために: $a = \log^F$

もちろん、これは純粋に形式的な変換ですが、対数のいくつかの特性を知っているれば、それはかなり役立ちます。特に、積型に基づく関手は和型で表現されることがわかり、和型関手は一般的には表現可能ではありません (例: リスト関手)。

最後に、表現可能関手は、同じことの二つの異なる実装を与えてくれることに注意してください — 一つは関数、もう一つはデータ構造です。それらにはまったく同じ内容があり — 同じキーを使って同じ値が取り出されます。それが私が話していた「同じさ」の感覚です。二つの自然に同型な関手は、その内容に関与する限りでは同一です。一方、二つの表現はしばしば異なって実装され、異なるパフォーマンス特性を持っています。メモ化はパフォーマンスの向上として使用され、大幅に実行時間を短縮することができます。同じ基礎となる計算の異なる表現を生成する能力は、実際には非常に貴重です。従って、驚くべきことに、パフォーマンスに関しては全く関心がないにも関わらず、

圈論は実用的な価値を持つ代替実装を探求するための豊富な機会を提供しています。

14.3 チャレンジ

1. ホム関手が C 内の恒等射を **Set** 内の対応する恒等関数に写すことを示してください。
2. **Maybe** が表現可能でないことを示してください。
3. **Reader** 関手は表現可能ですか？
4. **Stream** 表現を使って、引数を二乗する関数をメモ化してください。
5. **Stream** の **tabulate** と **index** が実際に互いの逆であることを示してください。(ヒント: 帰納法を使用します。)
6. 関手:

```
| Pair a = Pair a a
```

は表現可能です。それを表現する型を推測できますか？
tabulate と **index** を実装してください。

14.4 参考文献

- 表現可能関手についての Catsters ビデオ [representable functions^{*1}](#).

^{*1} <https://www.youtube.com/watch?v=4QgjKUzyrhM>

15

米田の補題



論のほとんどの構成は、数学の他のより具体的な領域からの結果の一般化です。積、余積、モノイド、指數などは、圏論が知られる前から知られていました。それらは数学の異なる分野で異なる名前で知られていたかもしれません。集合論におけるデカルト直積、順序論におけるミート、論理学における論理積 — これらはすべて圏論的積の抽象的なアイデアの具体的な例です。

この点で、米田の補題は他の数学の分野でほとんどまたは全く前例のない一般的な圏についての包括的な声明として際立っています。いくつかは、その最も近い類似物は群論の Cayley の定理です (すべての群はいくつかの集合の置換群と同型です)。

米田の補題の設定は、任意の圏 C と、 C から Set への関手 F とを合わせたものです。前の節で、いくつかの Set 値関手が表現可能である、つまりホム関手と同型であることを見ました。米田の補題は、すべての Set 値関手が自然変換を通じてホム関手から得られ、そのような変換を明示的に列挙すると言っています。

自然変換について話したとき、自然性条件がかなり制限的である可能性があると述べました。ある対象における自然変換のコンポーネントを定義すると、自然性はそれを射を通じて別の対象に「輸送」するのに十分強力かもしれません。ソースとターゲットの圏の対象間の射が多いほど、自然変換のコンポーネントを輸送するための制約が多くなります。 Set は非常に射が豊富な圏です。

米田の補題は、ホム関手と任意の他の関手 F との間の自然変換が、たった一点、つまり \mathbf{id}_a における α_a の単一の値を指定することによって完全に決定されることを教えてくれます。自然性条件から残りの自然変換がただちに従います。

だから、米田の補題に関わる二つの関手の間の自然性条件を見直しましょう。最初の関手はホム関手です。それは C の任意の対象 x を射の集合 $C(a, x)$ に写像します — a は C の固定された対象です。また、任意の射 f を $x \rightarrow y$ から $C(a, f)$ に写像することも見ました。

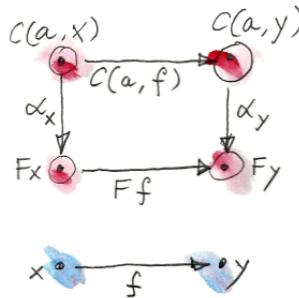
二番目の関手は任意の Set 値関手 F です。

これら二つの関手間の自然変換を α 呼びましょう。 Set で作業しているので、自然変換のコンポーネント、例えば α_x や α_y は单なる集

合間の通常の関数です:

$$\alpha_x : C(a, x) \rightarrow Fx$$

$$\alpha_y : C(a, y) \rightarrow Fy$$



そしてこれらが単なる関数であるので、私たちは特定の点でのそれらの値を見ることができます。しかし、集合 $C(a, x)$ の中の点とは何でしょうか？ ここが重要な観察です: 集合 $C(a, x)$ の中の全ての点もまた a から x への射 h です。

だから α に対する自然性の四角形:

$$\alpha_y \circ C(a, f) = Ff \circ \alpha_x$$

は、 h に作用するとき、点ごとに次のようにになります:

$$\alpha_y(C(a, f)h) = (Ff)(\alpha_x h)$$

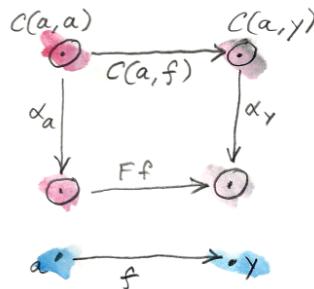
前の節から射 f に対するホム関手 $C(a, -)$ の作用は前合成として定義されたことを思い出してください:

$$C(a, f)h = f \circ h$$

それによって次のようにになります:

$$\alpha_y(f \circ h) = (Ff)(\alpha_x h)$$

この条件がどれほど強いかは、 $x = a$ の場合に特化することによって見ることができます。



その場合 h は a から a への射になります。少なくとも一つのそういう射、 $h = \mathbf{id}_a$ があることがわかっています。それを挿入しましょう:

$$\alpha_y f = (Ff)(\alpha_a \mathbf{id}_a)$$

ただちに何が起こったかに気づいてください: 左側は $C(a, y)$ の任意の要素 f に対する α_y の作用です。そしてそれは \mathbf{id}_a における α_a の单一の値によって完全に決定されます。私たちはそのような任意の値を選

ぶことができ、それは自然変換を生成します。 α_a の値は Fa の集合にありますから、 Fa の任意の点はいくつかの α を定義することができます。

逆に、 $C(a, -)$ から F への任意の自然変換 α が与えられたら、それを \mathbf{id}_a で評価して Fa の中の点を得ることができます。

これによって米田の補題が証明されました:

$C(a, -)$ から F への自然変換と Fa の要素との間には一対一の対応関係があります。

言い換えれば、

$$\text{Nat}(C(a, -), F) \cong Fa$$

あるいは、 C と Set の間の関手圏に対する表記 $[C, \text{Set}]$ を使えば、自然変換の集合はその圏のホム集合であり、次のように書くことができます:

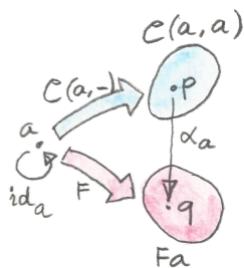
$$[C, \text{Set}](C(a, -), F) \cong Fa$$

この対応関係が実際には自然同型であることを後で説明します。

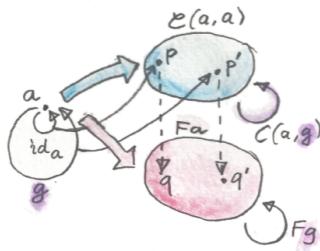
この結果についていくらか直感を得ましょう。最も驚くべきことは、全体の自然変換がただ一つの核となる点: \mathbf{id}_a でそれに割り当てる値から結晶化することです。それは自然性条件に従ってその点から広がります。それは C の Set への像を溢れさせます。だからまず、 $C(a, -)$ の下での C の像が何であるかを考えましょう。

まず a 自身の像から始めましょう。ホム関手 $C(a, -)$ の下で、 a は集合 $C(a, a)$ に写像されます。他方、関手 F の下では、それは集合 Fa に

写像されます。自然変換 α_a のコンポーネントは $C(a, a)$ から Fa へのいくつかの関数です。集合 $C(a, a)$ の中の一点、 id_a に対応する点に注目しましょう。それがただの集合の中の点であるという事実を強調するために、それを p と呼びましょう。コンポーネント α_a は p を Fa の中のいくつかの点 q に写像すべきです。任意の q の選択が一意の自然変換につながることを示します。



最初の主張は、一つの点 q の選択が残りの関数 α_a を一意に決定することです。確かに、 $C(a, a)$ の中の他の任意の点 p' を選びましょう。それは a から a へのいくつかの射 g に対応します。そして、ここで米田の補題の魔法が起こります: g は集合 $C(a, a)$ の中の点 p' として見ることができます。同時に、それは集合間の二つの関数を選択します。確かに、ホム関手の下で、射 g は関数 $C(a, g)$ に写像され、 F の下で Fg に写像されます。



今、 $C(a, g)$ が私たちの元の p 、すなわち \mathbf{id}_a に対応するものに作用することを考えましょう。それは前合成として定義されます、 $g \circ \mathbf{id}_a$ 、それは g に等しく、私たちの点 p' に対応します。だから射 g は、 p に作用すると p' を生成する関数に写像されます、それは g です。私たちの一回りしました！

今、 Fg が q に作用することを考えましょう。それは Fa の中のいくつかの点 q' です。自然性の四角形を完成させるために、 p' は α_a の下で q' に写像されなければなりません。私たちは任意の p' (任意の g) を選んで、 α_a の下でのその写像を導き出しました。関数 α_a はそうして完全に決定されます。

a に接続されている \mathbf{C} の任意の対象 x に対して、 α_x も一意に決定されるという第二の主張があります。推論は類似していますが、今度は 2 つの追加の集合、 $C(a, x)$ と Fx があり、 a から x への射 g はホム関手の下で次のように写像されます:

$$C(a, g) :: C(a, a) \rightarrow C(a, x)$$

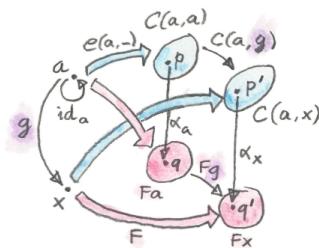
そして F の下では:

$$Fg :: Fa \rightarrow Fx$$

再び、 $C(a, g)$ が私たちの p に作用することは前合成によって与えられます: $g \circ \text{id}_a$ 、それは $C(a, x)$ の中の点 p' に対応します。自然性は α_x が p' に作用する値を決定します:

$$q' = (Fg)q$$

p' が任意だったので、関数 α_x 全体がそうして決定されます。



a に接続がない C の対象があった場合はどうでしょうか? それらはすべて $C(a, -)$ の下で单一の集合 – 空集合に写像されます。空集合は集合の圏の始対象です。それはこの集合から任意の他の集合への一意の関数があることを意味します。この関数を **absurd** と呼びます。だからここでも、自然変換のコンポーネントに対して選択の余地はありません: それは **absurd** 以外にはありません。

米田の補題を理解する一つの方法は、**Set** 値関手間の自然変換が関数の族であり、関数は一般的に情報を損失するものであることを認識

することです。関数は情報を崩壊させるかもしれませんし、その終域の一部だけを覆うかもしれません。損失がない唯一の関数は可逆なもの、つまり同型です。それに続いて、最良の構造を保持する **Set** 値関手は表現可能なものです。それらはホム関手か、ホム関手に自然に同型な関手です。任意の他の関手 F は、情報を失う変換を通じてホム関手から得られます。そのような変換は情報を失うだけでなく、関手 F の **Set** における像のごく一部しか覆わないかもしれません。

15.1 Haskell での米田

すでに Haskell でホム関手に出会いました、それは reader 関手の仮面をかぶっています:

```
type Reader a x = a -> x

type Reader[A, X] = A => X
```

Reader は射 (ここでは関数) を前合成によって写像します:

```
instance Functor (Reader a) where
    fmap f h = f . h

implicit def readerFunctor[A] = new Functor[Reader[A, ?]] {
    def fmap[X, B](f: X => B)(h: Reader[A, X]): Reader[A, B] =
        f compose h
```

```
}
```

米田の補題は、Reader 関手が任意の他の関手に自然に写像できると教えてくれます。

自然変換は多相的関数です。だから、関手 F が与えられたら、私たちは Reader 関手からそれに対する写像を持っています:

```
alpha :: forall x . (a -> x) -> F x

// In order to make a universal transformation,
// another type needs to be introduced.
// To read more about FunctionK (~>):
// typelevel.org/cats/datatypes/functionk.html
trait ~>[F[_], G[_]] {
  def apply[B](fa: F[B]): G[B]
}

// Kind Projector plugin provides
// a more concise syntax for type lambdas:
def alpha[A]: (A => ?) ~> F
```

通常、`forall` は省略可能ですが、自然変換のパラメトリック多相性を強調するために、私はそれを明示的に書くのが好きです。

米田の補題は、これらの自然変換が F a の要素と一一に対応していることを教えてくれます:

```
forall x . (a -> x) -> F x ≈ F a
```

この等式の右側は、私たちが通常データ構造と考えるものです。関手を一般化されたコンテナとしての解釈を覚えていますか？ $F a$ は a のコンテナです。しかし、左側は関数を引数として取る多相的関数です。米田の補題は、二つの表現が等価である、つまり同じ情報を含んでいることを教えてくれます。

これを言い換えると：私にこの型の多相的関数を与えてください：

```
alpha :: forall x . (a -> x) -> F x
```

```
def alpha[A]: (A => ?) ~> F
```

そして私は a のコンテナを生成します。トリックは、米田の補題の証明で使用したものと同じです：この関数を id で呼び出して、 $F a$ の要素を得ます：

```
alpha id :: F a
```

```
alpha(identity): F[A]
```

逆もまた真です： $F a$ の型の値が与えられた場合：

```
fa :: F a
```

```
def fa[A]: F[A]
```

誰かが正しい型の多相的関数を定義できます:

```
alpha h = fmap h fa
```

```
alpha(h) == fmap(h)(fa)
```

二つの表現間を簡単に行き来することができます。

複数の表現を持つ利点は、一方が他方よりも組み合わせやすいか、あるいはいくつかの応用においてより効率的である可能性があることです。

この原理の最も単純な例は、しばしばコンパイラ開発に使用されるコード変換です: 継続渡しスタイルまたは CPS です。それは、恒等関手に米田の補題を適用した最も単純な応用です。F を恒等関手で置き換えると:

```
forall r . (a -> r) -> r ≡ a
```

この公式の解釈は、任意の型 a が a を受け取る「ハンドラー」を取る関数に置き換えられるということです。ハンドラーは a を受け入れて残りの計算を行う関数です。(型 r は通常、何らかの種類の状態コードをカプセル化します。)

このプログラミングスタイルは、UI、非同期システム、および並行プログラミングで非常に一般的です。CPS の欠点は、制御の反転が含

まれることです。コードはプロデューサーとコンシューマー（ハンドラー）の間で分割され、容易に組み合わせることができます。状態フルハンドラーが交錯するスペゲッティコードの悪夢に慣れている人は誰でも、Web プログラミングをある程度以上行ったことがあるでしょう。後で見るように、関手とモナドの賢明な使用は、CPS の組成特性のいくつかを復元することができます。

15.2 余米田

いつものように、射の方向を反転することによってボーナス構造を得ます。米田の補題は、逆圏 \mathbf{C}^{op} に適用され、反変関手間の写像を与えます。

同等に、私たちは、私たちのホム関手の代わりにターゲット対象を固定することによって、余米田の補題を導出することができます。私たちは \mathbf{C} から \mathbf{Set} への反変ホム関手 $\mathbf{C}(-, a)$ を得ます。余米田の補題の反変版は、この関手から任意の他の反変関手 F への自然変換と Fa の集合の要素との間に一对一の対応を確立します:

$$\mathbf{Nat}(\mathbf{C}(-, a), F) \cong Fa$$

これが Haskell 版の余米田の補題です:

```
forall x . (x -> a) -> F x ≡ F a
```

いくつかの文献では、反変版が米田の補題と呼ばれていることに注意してください。

15.3 チャレンジ

1. Haskell での米田同型を形成する二つの関数 `phi` と `psi` が互いに逆であることを示してください。

```
phi :: (forall x . (a -> x) -> F x) -> F a  
phi alpha = alpha id
```

```
psi :: F a -> (forall x . (a -> x) -> F x)  
psi fa h = fmap h fa
```

2. 対象はあるが、恒等射以外の射がない離散圏に対して、米田の補題はどのように機能しますか？
3. 単位のリスト `[]` は、その長さ以外の情報を含みません。したがって、データ型として、それは整数のエンコーディングと見なすことができます。空リストはゼロを、一つの要素のリスト `[()` (値であり型ではありません) は一を、そして以下同様です。リスト関手に対する米田の補題を使用して、このデータ型の別の表現を構成してください。

15.4 参考文献

1. Catsters^{*1}のビデオ。

^{*1} <https://www.youtube.com/watch?v=TLMxHB19khE>

16

米田埋め込み

これまでに、圏 \mathbf{C} の対象 a を固定した場合、写像 $\mathbf{C}(a, -)$ は \mathbf{C} から \mathbf{Set} への(共変)関手であることを見てきました。

$$x \rightarrow \mathbf{C}(a, x)$$

(終域は \mathbf{Set} です、なぜならホム集合 $\mathbf{C}(a, x)$ は集合だからです。) この写像をホム関手と呼びます。 $-$ 以前に射に対するその作用も定義しました。

さて、この写像において a を変化させてみましょう。新しい写像が得られ、それは任意の a に対してホム関手 $\mathbf{C}(a, -)$ を割り当てます。

$$a \rightarrow \mathbf{C}(a, -)$$

これは対象から関手への写像であり、つまり C の対象から関手圏の対象への写像です（関手圏については自然変換の節を参照してください）。関手圏 C から Set への記法として $[C, \text{Set}]$ を使いましょう。また、ホム関手は典型的な表現可能関手であることを覚えているかもしれません。

二つの圏の間に対象の写像がある場合、その写像が関手であるかどうか、つまり一方の圏の射を他方の圏の射に持ち上げができるかどうかを自然に考えます。 C の射は $C(a, b)$ の要素ですが、関手圏 $[C, \text{Set}]$ の射は自然変換です。したがって、私たちは射を自然変換に写す写像を探しています。

射 $f :: a \rightarrow b$ に対応する自然変換を見つけられるか見てみましょう。まず、 a と b が何に写されるかを見てみましょう。それらは二つの関手、 $C(a, -)$ と $C(b, -)$ に写されます。この二つの関手の間の自然変換が必要です。

ここでコツがあります。米田の補題を使います：

$$[C, \text{Set}](C(a, -), F) \cong Fa$$

そして一般的な F をホム関手 $C(b, -)$ に置き換えます。私たちは得ます：

$$[C, \text{Set}](C(a, -), C(b, -)) \cong C(b, a)$$

これはまさに私たちが探していた二つのホム関手間の自然変換ですが、少し捻りがあります。自然変換と射— $C(b, a)$ の要素—との間には

「逆」方向への写像があります。しかし、それは問題ありません。これは私たちが見ている関手が反変であることを意味します。

実際、私たちは想定以上のものを得ました。 \mathbf{C} から $[\mathbf{C}, \mathbf{Set}]$ への写像は反変関手だけでなく、**充満忠実関手**です。充満性と忠実性は、関手がホム集合をどのように写像するかを記述する関手の性質です。

忠実関手はホム集合において**単射**であり、つまりそれは異なる射を異なる射に写像します。言い換えれば、それは射をまとめてしまうことはありません。

充満関手はホム集合において**全射**であり、つまりそれは一つのホム集合を別のホム集合に上へ写像し、後者を完全に覆います。

充満忠実関手 F はホム集合上の**全単射**です—両方の集合の全ての要素の一対一の対応です。源の圏 \mathbf{C} の任意の対象のペア a と b に対して、 $\mathbf{C}(a, b)$ と $\mathbf{D}(Fa, Fb)$ の間には全単射があります。ここで \mathbf{D} は F の対象圏です(この場合、関手圏 $[\mathbf{C}, \mathbf{Set}]$ です)。これは F が**対象**上の全単射であるとは言いません。 \mathbf{D} には F の像にない対象が存在するかもしれませんし、そのような対象のホム集合については何も言えません。

16.1 埋め込み

私たちがちょうど記述した(反変)関手、つまり \mathbf{C} の対象を $[\mathbf{C}, \mathbf{Set}]$ の関手に写す関手:

$$a \rightarrow \mathbf{C}(a, -)$$

は米田埋め込みを定義します。それは圏 C (厳密には反変性のために C^{op}) を関手圏 $[C, Set]$ 内に埋め込むものです。それは C の対象を関手に写すだけでなく、それらの間の全ての接続を忠実に保存します。

これは非常に有用な結果です。なぜなら、数学者は関手の圏、特に終域が **Set** である関手について多くのことを知っているからです。私たちは任意の圏 C について、それを関手圏に埋め込むことによって多くの洞察を得ることができます。

もちろん、米田埋め込みには双対バージョンがあり、時に余米田埋め込みと呼ばれることがあります。私たちが各ホム集合の対象 (ソース対象ではなく) を固定することから始めることができたことに注意してください、 $C(-, a)$ 。それは私たちに反変ホム関手を与えます。 C から **Set** への反変関手は私たちがよく知る前層 (例えば、**極限**と**余極限**を参照してください) です。余米田埋め込みは圏 C を前層の圏に埋め込むことを定義します。射に対するその作用は以下の通りです:

$$[C, Set](C(-, a), C(-, b)) \cong C(a, b)$$

再び、前層の圏について多くのことが知られているので、任意の圏をそこに埋め込むことができる大きな利点です。

16.2 Haskell への応用

Haskell では、米田埋め込みは reader 関手間の自然変換と、それは逆方向の関数との間の同型として表現できます:

```
forall x. (a -> x) -> (b -> x) ≡ b -> a
```

(reader 関手は $((\rightarrow) a)$ と同等です。)

この恒等式の左辺は、 a から x への関数と b の型の値を与えられたときに、 x の型の値を生成することができる多相的関数です（私は関数 $b \rightarrow x$ の周りの括弧を落としています—非 Curry 化しています）。これがすべての x に対して行える唯一の方法は、関数が b を a に変換する方法を密かに知っている場合です。

そのようなコンバーター、 $btoa$ がある場合、左辺を `fromY` と呼び、次のように定義することができます：

```
fromY :: (a -> x) -> b -> x
fromY f b = f (btoa b)
```

```
// In order to make a universal transformation,
// another type needs to be introduced.
// To read more about FunctionK (~>):
// typelevel.org/cats/datatypes/functionk.html
trait ~>[F[_], G[_]] {
  def apply[X](fa: F[X]): G[X]
}

def fromY[A, B]: (A => ?) ~> (B => ?) = new ~>[A => ?, B => ?] {
  def apply[X](f: A => X): B => X =
    b => f(btoa(b))
}
```

逆に、関数 `fromY` が与えられれば、`identity` を呼び出すことによってコンバーターを回復することができます:

```
fromY id :: b -> a
```

```
fromY(identity): B => A
```

これは型 `fromY` と `btoa` の関数の間の全単射を確立します。

この同型を見る別の方法は、それが `b` から `a` への関数の CPS エンコーディングであるということです。引数 `a -> x` は継続(ハンドラー)です。結果は `b` から `x` への関数で、`b` の型の値が与えられたときに、エンコードされている関数を事前に合成して継続を実行します。

米田埋め込みはまた、特にそれが `Control.Lens` ライブラリからのレンズの非常に有用な表現^{*1}を提供することによって、Haskell のデータ構造の代替表現を説明します。

16.3 前順序の例

この例は Robert Harper によって提案されました。それは前順序によって定義される圏への米田埋め込みの適用です。前順序は \leq (以下)として伝統的に書かれる要素間の順序関係を持つ集合です。前順序における「pre」は、関係が推移的で反射的であることが求められるのに

^{*1} <https://bartoszmilewski.com/2015/07/13/from-lenses-to-yoneda-embedding/>

対し、必ずしも反対称的でないということを意味します（したがって、サイクルが可能です）。

前順序関係を持つ集合は圏を生み出します。対象はこの集合の要素です。対象 a から b への射は、対象が比較できないか、 $a \leq b$ でない場合には存在しないか、または $a \leq b$ の場合に存在し、 a から b へと向かいます。一つの対象から別の対象への射は決して一つ以上存在しません。従って、このような圏の任意のホム集合は空集合か単集合です。このような圏を薄いと呼びます。

$a \leq b$ かつ $b \leq c$ ならば $a \leq c$ であることから、射は合成可能ですし、合成は結合的です。また、各要素は自分自身に等しい（関係の反射性）ので、恒等射も存在します。

前順序圏に余米田埋め込みを適用することができます。特に、射の作用に興味があります。

$$[\mathbf{C}, \mathbf{Set}](\mathbf{C}(-, a), \mathbf{C}(-, b)) \cong \mathbf{C}(a, b)$$

右側のホム集合は $a \leq b$ の場合に限り空でないです — その場合は単集合です。従って、 $a \leq b$ ならば、左側には一つの自然変換が存在します。そうでない場合は自然変換は存在しません。

前順序におけるホム関手間の自然変換はどのようなものでしょうか？ それは集合 $\mathbf{C}(-, a)$ と $\mathbf{C}(-, b)$ の間の関数の族です。前順序では、これらの各集合は空か単集合です。利用可能な関数の種類を見てみましょう。

空集合から自身への関数(空集合に対する恒等関数)、空集合から単集合への関数(absurd、それは定義される必要があるが空集合の要素はないので何もしません)、単集合から自身への関数(単集合に対する恒等関数)があります。唯一許されていない組み合わせは、単集合から空集合への写像です(そのような関数が单一の要素に対してどのような値を取るか?)。

したがって、自然変換は決して単ホム集合を空ホム集合に接続しません。言い換えれば、もし $x \leq a$ (単ホム集合 $C(x, a)$)ならば、 $C(x, b)$ は空であってはなりません。非空の $C(x, b)$ は $x \leq b$ であることを意味します。従って、問題の自然変換の存在は、すべての x に対して、 $x \leq a$ ならば $x \leq b$ が必要です。

$$\text{すべての } x \text{ に対して}, x \leq a \Rightarrow x \leq b$$

一方、余米田はこの自然変換の存在が $C(a, b)$ が空でないこと、つまり $a \leq b$ であることと同等であると述べています。これらを合わせると、以下のことが得られます:

$$a \leq b \text{ は } \text{すべての } x \text{ に対して}, x \leq a \Rightarrow x \leq b \text{ と同等です}$$

この結果には直接到達することもできますが、米田埋め込みを通じてこの結果に到達することは、はるかに興奮します。

16.4 自然性

米田の補題は、自然変換の集合と **Set** 内の対象との間の同型を確立します。自然変換は関手圏 $[C, Set]$ 内の射です。任意の二つの関手間の自然変換の集合は、その圏のホム集合です。米田の補題は次の同型です:

$$[C, Set](C(a, -), F) \cong Fa$$

この同型は、 F と a の両方について自然です。つまり、 $[C, Set] \times C$ という積圏からのペア (F, a) について自然です。ここで、 F を関手圏の対象として扱っていることに注目してください。

これが意味することを少し考えてみましょう。自然同型は、二つの関手間の可逆な自然変換です。そして確かに、私たちの同型の右側は関手です。それは $[C, Set] \times C$ から **Set** への関手で、ペア (F, a) に対する作用は集合-関手 F が対象 a で評価された結果です。これは評価関手と呼ばれます。

左側もまた関手で、 (F, a) を自然変換の集合 $[C, Set](C(a, -), F)$ に写像します。

これらが本当に関手であることを示すためには、射に対するそれらの作用も定義する必要があります。しかし、ペア (F, a) と (G, b) の間の射とは何でしょうか？ それは射のペア、 (Φ, f) です。一つ目は関手間の射-自然変換、二つ目は C 内の通常の射です。

評価関手はこのペア (Φ, f) を取り、二つの集合 Fa と Gb の間の関数に写像します。私たちは Φ の a でのコンポーネント (Fa から Ga への

写像) と、 G によって持ち上げられた射 f から、容易にそのような関数を構成することができます:

$$(Gf) \circ \Phi_a$$

Φ の自然性により、これは以下と同じです:

$$\Phi_b \circ (Ff)$$

私はここでは同型の全体の自然性を証明しませんが、関手と自然変換から構成されているので、それが誤る余地はありません。

16.5 チャレンジ

1. Haskell で余米田埋め込みを表現せよ。
2. `fromY` と `btoa` の間に確立した全単射が同型であることを示せ(二つの写像は互いに逆である)。
3. モノイドの米田埋め込みを詳しく調べよ。モノイドの单一の対象に対応する関手は何か？ モノイドの射に対応する自然変換は何か？
4. 前順序に対する共変米田埋め込みの応用は何か？ (Gershon Bazerman による提案)
5. 米田埋め込みを使って、任意の関手圏 $[C, D]$ を関手圏 $[[C, D], Set]$ に埋め込む方法を考えよ。この場合、射(つまり自然変換)に対する作用を図り出せ。

第3部

17

すべては射に関することです

も しあなたが射についてであると納得していないならば、私の仕事は不十分です。次の話題は随伴で、これはホム集合の同型の観点から定義されますから、ホム集合の構成要素についての私たちの直感を見直すのは理にかなっています。また、随伴は我々が以前に学んだ多くの構成法を記述するための一般的な言語を提供しますから、それらを見直すのも役立つでしょう。

17.1 関手

まず、関手を射の写像として考えるべきです – Haskell の `Functor` 型クラスの定義で強調されている視点です、これは `fmap` を中心に展

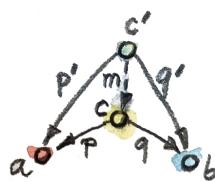
開されます。もちろん、関手は対象 - 射の端点 - も写像します。さもなければ合成を保つことについて話すことができません。対象はどの射のペアが合成可能かを教えてくれます。一つの射の目標が他の射の源と等しくなければならない - もし彼らが合成されるならば。ですから、射の合成が持ち上げられた射の合成に写像されることを望むならば、その端点の写像はほとんど決定されています。

17.2 可換図式

射の多くの性質は可換図式の観点から表現されます。特定の射が他の射の合成として複数の方法で記述される場合、可換図式を持っています。

特に、可換図式はほぼ全ての普遍構成の基礎を形成します(始対象と終対象の顕著な例外を除いて)。積、余積、様々な他の(余)極限、指數対象、自由モノイドなどの定義でこれを見てきました。

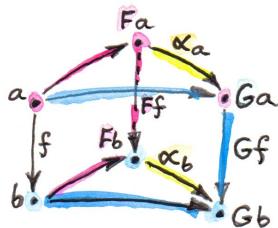
積は普遍構成の簡単な例です。我々は二つの対象 a と b を選び、それらの積である普遍性を持つ対象 c が存在するかどうかと、一対の射 p と q を見ます。



積は極限の特別なケースです。極限は錐の観点から定義されます。一般的な錐は可換図式から構成されます。これらの図式の可換性は、関手の写像に適切な自然性条件に置き換えられます。この方法で可換性は自然変換の高水準言語の組み立て言語の役割に還元されます。

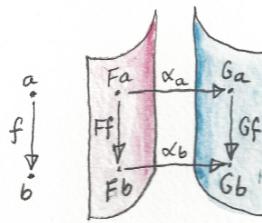
17.3 自然変換

一般的に、射から可換な四角形への写像が必要な場合に、自然変換は非常に便利です。自然性四角形の二つの対立する側面は、二つの関手 F と G の下でのいくつかの射 f の写像です。他の側面は自然変換のコンポーネントです（これもまた射です）。



自然性は、あなたが「隣接する」コンポーネントに移動するとき（隣接するとは、射によって接続されていることを意味します）、圏や関手の構造に反することがないことを意味します。自然変換のコンポーネントを使用して対象間の隙間を橋渡ししてから、関手を使用してその隣人にジャンプするのか、またはその逆かは問題ではありません。二

つの方向は直交しています。自然変換はあなたを左右に移動させ、関手はあなたを上下または前後に移動させます。目標圏内の関手の像をシートとして視覚化することができます。自然変換は、F に対応するシートから、G に対応する別のシートへと写像します。



Haskell でこの直交性の例を見てきました。そこでは、関手の作用は容器の内容を変更せずに形を変えずに、自然変換は触れられていない内容を異なる容器に再梱包します。これらの操作の順序は問題ではありません。

極限の定義における錐は自然変換に置き換えられています。自然性は、すべての錐の側面が交通することを保証します。それでも、錐間の写像として極限は定義されています。これらの写像もまた交通条件を満たさなければなりません。(例えば、積の定義における三角形は交通しなければなりません。)

これらの条件もまた自然性に置き換えられるかもしれません。あなたはおそらく、普遍的な錐、または極限が(反変) ホム関手との間の自

然変換として定義されることを思い出でましょう:

$$F :: c \rightarrow \mathbf{C}(c, \mathbf{Lim}D)$$

および、(また反変的な) \mathbf{C} 内の対象を自然変換である錐に写像する関手:

$$G :: c \rightarrow \mathbf{Nat}(\Delta_c, D)$$

ここで、 Δ_c は定数関手であり、 D は \mathbf{C} 内の図式を定義する関手です。両方の関手 F と G は、 \mathbf{C} 内の射に対して明確に定義された動作を持っています。この特定の自然変換は、 F と G の間の同型です。

17.4 自然同型

自然同型 – すべてのコンポーネントが可逆な自然変換 – は圏論が「二つのものが同じである」という方法です。このような変換のコンポーネントは、対象間の同型 – 逆を持つ射 – でなければなりません。関手の像をシートとして視覚化する場合、自然同型はこれらのシート間の一対一の可逆写像です。

17.5 ホム集合

しかし、射とは何ですか？ 射は対象よりも多くの構造を持っています：対象とは異なり、射には二つの端があります。しかし、源と目標対象を固定すると、二つの間の射は退屈な集合を形成します（少なくとも

も局所的に小さい圏について)。私たちはこの集合の要素に f や g のような名前を付けて、一つを別のものと区別することができます – しかし、それは実際にそれらを異なるものにしているのは何でしょうか？

与えられたホム集合内の射の本質的な違いは、他の射との合成方法にあります(隣接するホム集合から)。射 h が存在して、 f との合成(前または後)が g と異なる場合、例えば:

$$h \circ f \neq h \circ g$$

その場合、私たちは直接 f と g の違いを「観察」することができます。しかし、違いが直接観察できない場合でも、関手を使ってホム集合にズームインすることができるかもしれません。関手 F は二つの射を異なる射に写像するかもしれません:

$$Ff \neq Fg$$

より豊かな圏で、隣接するホム集合がより多くの解像度を提供するところでの、例えば、

$$h' \circ Ff \neq h' \circ Fg$$

ここで h' は F の像の中にはないです。

17.6 ホム集合同型

多くの圏論的構成はホム集合間の同型に依存しています。しかし、ホム集合は单なる集合なので、それらの間の单なる同型はあまり多く

を語りません。有限集合に対しては、同型はそれらが同じ数の要素を持っていることを言います。集合が無限の場合、その濃度が同じでなければなりません。しかし、ホム集合の意味のある同型は合成を考慮に入れなければなりません。そして、合成は一つ以上のホム集合を関与させます。我々は、合成との互換性条件を課すことで、全てのホム集合の集合をまたがる同型を定義する必要があります。そして、**自然同型**がまさにその要求に合っています。

しかし、ホム集合の自然同型とは何でしょうか？自然性は、関手間の写像の性質であり、集合の性質ではありません。だから、私たちは実際にはホム集合に価値のある関手間の自然同型について話しています。これらの関手は、单なる集合価値の関手以上のものです。射の動作は、適切なホム関手によって誘導されます。射は、ホム関手によって前または後の合成(関手の共変性に応じて)を使って典型的に写像されます。

米田埋め込みは、このような同型の一例です。それは \mathbf{C} 内のホム集合を関手圏内のホム集合に写像し、それは自然です。米田埋め込みの一つの関手は \mathbf{C} 内のホム関手であり、もう一つは対象をホム集合間の自然変換の集合に写像します。

極限の定義もまたホム集合間の自然同型です(2番目もまた関手圏内で)：

$$\mathbf{C}(c, \mathbf{Lim}D) \simeq \mathbf{Nat}(\Delta_c, D)$$

実は、指数対象の構成や自由モノイドの構成も、ホム集合間の自然同型として書き直すことができます。

これは偶然の一致ではありません – 次に見るように、これらは随伴の異なる例にすぎません。随伴はホム集合の自然同型として定義されます。

17.7 ホム集合の非対称性

随伴を理解する上でさらに一つの観察が役立ちます。ホム集合は一般に非対称です。ホム集合 $C(a, b)$ はしばしばホム集合 $C(b, a)$ と非常に異なります。この非対称性の究極の実証は半順序としての圏です。半順序では、 a が b 以下である場合にのみ、 a から b への射が存在します。 a と b が異なる場合、逆方向、つまり b から a への射は存在しません。従って、ホム集合 $C(a, b)$ が空でない場合、つまり単集合である場合、 $a = b$ でない限り $C(b, a)$ は空でなければなりません。この圏では射には明確な一方向の流れがあります。

前順序は、必ずしも反対称ではない関係に基づいていますが、「主に」方向性があります、ただし、時折サイクルが発生します。任意の圏を前順序の一般化として考えると便利です。

前順序は薄い圏です – すべてのホム集合は単集合または空です。一般的な圏を「厚い」前順序として視覚化することができます。

17.8 チャレンジ

1. 自然性条件のいくつかの変形例を考え、適切な図式を描いてください。例えば、関手 F または G が $f :: a \rightarrow b$ の端点である対象 a と b の両方を同じ対象に写像する場合、例えば $Fa = Fb$ または $Ga = Gb$ の場合はどうなりますか？(この方法で錐または余錐が得られます。) 次に、 $Fa = Ga$ または $Fb = Gb$ の場合を考えてください。最後に、自身にループする射、つまり $f :: a \rightarrow a$ から始める場合はどうなりますか？

18

随伴

数 学では、あるものが他のものと似ているということを様々な方法で表現します。最も厳密なのは等価性です。二つのものが等しいとは、一方を他方と区別できないことを意味します。一方をどんな想像上の文脈においても他方と置き換えることができます。例えば、可換図式について話すたびに、射の**等価性**を使ったことに気付きましたか？ それは射が集合（ホム集合）を形成し、集合の要素は等価性で比較できるからです。

しかし、等価性はしばしば強すぎます。実際に等しくないけれども、すべての意図と目的において同じであるものの多くの例があります。例えば、ペア型 (`Bool`, `Char`) は厳密には (`Char`, `Bool`) と等しくあ

りませんが、同じ情報を含んでいることがわかります。この概念は、二つの型の間の同型、すなわち可逆な射によって最もよく捉えられます。それが射であるため、構造を保持し、「iso」であることは、どちら側から始めても同じ場所に着地する往復の一部であることを意味します。ペアの場合、この同型は **swap** と呼ばれます：

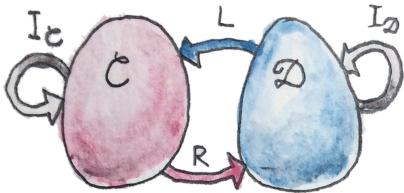
```
swap :: (a, b) -> (b, a)
swap (a, b) = (b, a)
```

```
def swap[A, B]: ((A, B)) => (B, A) = {
    case (a, b) => (b, a)
}
```

swap はたまたま自己逆です。

18.1 随伴と単位/余単位の対

圏が同型であると言うとき、それは圏の間の写像、つまり関手の観点で表現されます。二つの圏 **C** と **D** が同型である場合、**C** から **D** への関手 **R**（「右」）が存在し、それが可逆であると言えます。言い換えれば、**D** から **C** に戻る別の関手 **L**（「左」）が存在し、**R** と合成したものが恒等関手 **I** と等しくなります。可能な合成は **R**。**L** と **L**。**R** の二つで、可能な恒等関手は **C** に一つ、**D** にもう一つあります。



しかし、ここで厄介な部分があります: 二つの関手が等しいとはどういう意味ですか？ 以下の等式はどういう意味でしょうか：

$$R \circ L = I_D$$

あるいはこれは：

$$L \circ R = I_C$$

関手の等価性を対象の等価性の観点で定義するのは合理的でしょう。等しい対象に作用する二つの関手は等しい対象を生成すべきです。しかし、一般には、任意の圏における対象の等価性の概念を持っていません。それは単に定義の一部ではありません。（「等価性が本当に何であるか」というこの問題の深淵に深く踏み込むと、私たちはホモトピー型理論に行き着くでしょう。）

関手は圏の圏における射であるため、等価性比較が可能であるべきだと主張するかもしれません。そして確かに、私たちが小さい圏について話している限り、対象が集合を形成するところでは、集合の要素の等価性を使って対象を等価性比較することができます。

しかし、Cat は実際には 2-圏です。2-圏のホム集合には追加の構造があります—1-射間で作用する 2-射があります。Cat では、1-射は関

手で、2-射は自然変換です。ですから、関手について話すときに等価性の代わりに自然同型を考慮する方が自然（このダジャレは避けられません！）です。

したがって、圏の同型の代わりに、二つの圏 C と D が同値であると考える方が一般的です。私たちは二つの圏の間を往復する関手を見つけることができ、その合成（どちらの方向でも）が恒等関手と自然に同型です。つまり、合成 $R \circ L$ と恒等関手 I_D の間には二方向の自然変換があり、もう一方は $L \circ R$ と恒等関手 I_C の間にあります。

随伴は同値よりもさらに弱いです。それは二つの関手の合成が恒等関手に同型である必要はなく、 I_D から $R \circ L$ への一方向の自然変換と、 $L \circ R$ から I_C へのもう一方の存在を要求するだけです。これら二つの自然変換のシグネチャはこうです：

$$\begin{aligned}\eta &:: I_D \rightarrow R \circ L \\ \varepsilon &:: L \circ R \rightarrow I_C\end{aligned}$$

η は単位と呼ばれ、 ε は随伴の余単位です。

これら二つの定義の間の非対称性に注意してください。一般に、私たちは残りの二つの写像を持っていません：

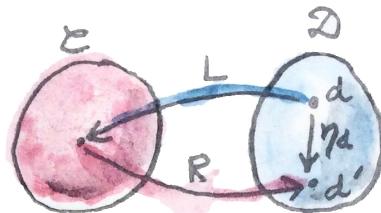
$$\begin{aligned}R \circ L \rightarrow I_D &\quad \text{必ずしもではありません} \\ I_C \rightarrow L \circ R &\quad \text{必ずしもではありません}\end{aligned}$$

この非対称性のために、関手 L は関手 R の左随伴と呼ばれ、関手 R は L の右随伴です。（もちろん、左と右はあなたが図を特定の方法で描く場合にのみ意味があります。）

随伴のコンパクトな表記は:

$$L \dashv R$$

随伴をより深く理解するために、単位と余単位をもっと詳しく分析しましょう。

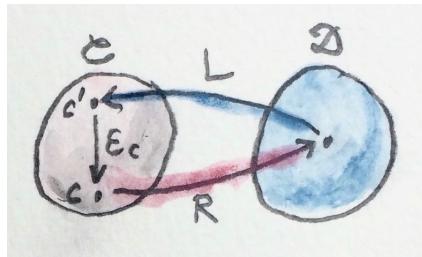


まず単位から始めましょう。それは自然変換なので、射の族です。**D** の対象 d を与えられたとき、 η のコンポーネントは Id 、つまり d 自身と $(R \circ L)d$ の間の射です。図では、 d' と呼ばれています:

$$\eta_d :: d \rightarrow (R \circ L)d$$

$R \circ L$ の合成は **D** の自己関手です。

この方程式は、任意の対象 d を **D** で始点として選び、 $R \circ L$ の往復関手を使って目標対象 d' を選ぶことができる教えてくれます。そして私たちは矢印 - 射 η_d - を目標に向けて撃ちます。



同様に、余単位 ε のコンポーネントは次のように記述できます:

$$\varepsilon_c :: (L \circ R)c \rightarrow c$$

これは、任意の対象 c を C で目標として選び、往復関手 $L \circ R$ を使って出発点 $c' = (L \circ R)c$ を選ぶことができるることを示しています。そして私たちは矢印一射 ε_c を出発点から目標へ撃ちます。

単位と余単位を見るもう一つの方法は、単位が D 上の恒等関手のどこにでも合成 $R \circ L$ を導入することを可能にし、余単位が C 上の恒等関手と置き換えることで合成 $L \circ R$ を排除することを可能にするということです。これは、導入に續いて排除が何も変えないことを確実にするいくつかの「明白な」整合性条件につながります:

$$L = L \circ I_D \rightarrow L \circ R \circ L \rightarrow I_C \circ L = L$$

$$R = I_D \circ R \rightarrow R \circ L \circ R \rightarrow R \circ I_C = R$$

これらは三角恒等式と呼ばれるもので、以下の図式を可換にします:

$$\begin{array}{ccc}
 L & \xrightarrow{L \circ \eta} & L \circ R \circ L \\
 & \searrow \quad \swarrow & \downarrow \epsilon \circ L \\
 & & L
 \end{array}
 \qquad
 \begin{array}{ccc}
 R & \xrightarrow{\eta \circ R} & R \circ L \circ R \\
 & \searrow \quad \swarrow & \downarrow R \circ \epsilon \\
 & & R
 \end{array}$$

これらは関手圏の図式です: 射は自然変換であり、その合成は自然変換の水平合成です。コンポーネントでこれらの恒等式を表現すると、次のようにになります:

$$\begin{aligned}
 \epsilon_{Ld} \circ L\eta_d &= \mathbf{id}_{Ld} \\
 R\epsilon_c \circ \eta_{Rc} &= \mathbf{id}_{Rc}
 \end{aligned}$$

単位と余単位は Haskell で異なる名前でよく見かけます。単位は `return` (または `Applicative` の定義における `pure`) として知られています:

```

return :: d -> m d

// return is a keyword in Scala
def pure[D]: D => M[D]

```

余単位は `extract` として知られています:

```
extract :: w c -> c
```

```
def extract[C]: W[C] => C
```

ここで、`m` は $R \circ L$ に対応する(自己)関手で、`w` は $L \circ R$ に対応する `v`(自己)関手です。後ほど見るように、これらはモナドと余モナドの定義の一部です。

自己関手をコンテナと考えると、単位(または `return`)は任意の型の値の周りにデフォルトのボックスを作成する多相的関数です。余単位(または `extract`)はその逆で、コンテナから单一の値を取り出すか生成します。

後ほど見るように、随伴のペアごとにモナドと余モナドが定義されます。逆に、すべてのモナドまたは余モナドは随伴のペアに分解することができます—この分解は一意ではありませんが。

Haskell では、私たちはモナドを頻繁に使用しますが、通常それらを随伴のペアに分解することは稀です。主に、それらの関手が通常私たちを `Hask` の外に連れて行くからです。

しかし、Haskell では自己関手の随伴を定義することができます。こちらが `Data.Functor.Adjunction` からの定義の一部です:

```
class (Functor f, Representable u) =>
    Adjunction f u | f -> u, u -> f where
    unit :: a -> u (f a)
    counit :: f (u a) -> a
```

```
abstract class Adjunction[F[_], U[_]](implicit val F: Functor[F], val U: Representable[U]){

  def unit[A](a: A): U[F[A]]

  def counit[A](a: F[U[A]]): A
}
```

この定義はいくつかの説明が必要です。まず第一に、それは複数のパラメータ型クラスを記述しています一二つのパラメータは f と u です。それはこれら二つの型コンストラクタ間の関係として `Adjunction` を確立します。

垂直バーの後の追加条件は機能依存性を指定します。例えば、 $f \rightarrow u$ は u が f によって決定されることを意味します（ここでは型コンストラクタ上の関数としての関係です）。逆に、 $u \rightarrow f$ は、 u を知つていれば、 f が一意に決定されることを意味します。

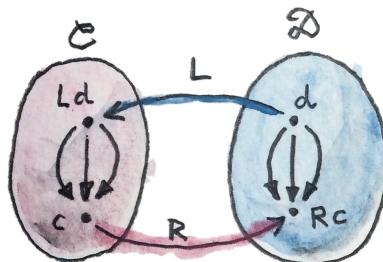
18.2 随伴とホム集合

随伴の別の同等の定義は、ホム集合の自然同型の観点からです。この定義は、これまでに検討してきた普遍構成とうまく結びついています。何か特有の射が、何かの構成を因数分解するという声明を聞くた

びに、それをある集合からホム集合への写像として考えるべきです。 「特有の射を選ぶ」ということの意味です。

さらに、因数分解はしばしば自然変換の観点で記述されます。因数分解には可換図式が関与します—ある射が二つの射(因数)の合成に等しいです。自然変換は射を可換図式に写像します。したがって、普遍構成では、私たちは射から可換図式へ、そして特有の射へ行きます。私たちは射から射へ、あるいは一つのホム集合から別のホム集合へ(通常は異なる圏内)の写像を終えます。この写像が可逆であり、それがすべてのホム集合に自然に拡張できる場合、私たちは随伴を持っています。

普遍構成と随伴の主な違いは、後者が全てのホム集合に対して大局的に定義されるということです。例えば、普遍構成を使用して、それがその圏の他のどのペアの対象に対しても存在しないかもしれないある選択された対象の積を定義することができます。すぐに見るように、任意のペアの対象の積が圏内に存在する場合、それはまた随伴を通じて定義することもできます。



ここに、ホム集合を使用して随伴を定義するための別の方法があります。前と同様に、私たちは二つの関手 $L :: \mathbf{D} \rightarrow \mathbf{C}$ と $R :: \mathbf{C} \rightarrow \mathbf{D}$ を持っています。私たちは二つの任意の対象を選びます: \mathbf{D} の出発対象 d と \mathbf{C} の目標対象 c です。 L を使って出発対象 d を \mathbf{C} に写像することができます。今、私たちは \mathbf{C} 内の二つの対象、 Ld と c を持っています。彼らはホム集合を定義します:

$$\mathbf{C}(Ld, c)$$

同様に、 R を使って目標対象 c を写像することができます。今、私たちは \mathbf{D} 内の二つの対象、 d と Rc を持っています。彼らもホム集合を定義します:

$$\mathbf{D}(d, Rc)$$

私たちは、 L が R の左随伴であると言います、もしホム集合の同型が存在すれば:

$$\mathbf{C}(Ld, c) \cong \mathbf{D}(d, Rc)$$

これは、 d と c の両方において自然です。自然というのは、出発点 d を \mathbf{D} を通してスムーズに変化させることができること、および目標点 c を \mathbf{C} を通して変化させることができることを意味します。より正確には、私たちは以下の二つの(共変)関手間の自然変換 φ を持っています。これらは \mathbf{C} から \mathbf{Set} への関手の対象への作用です:

$$\begin{aligned} c &\rightarrow \mathbf{C}(Ld, c) \\ c &\rightarrow \mathbf{D}(d, Rc) \end{aligned}$$

もう一方の自然変換、 ψ は、以下の(反変)関手間で作用します:

$$\begin{aligned}d &\rightarrow \mathbf{C}(Ld, c) \\d &\rightarrow \mathbf{D}(d, Rc)\end{aligned}$$

両方の自然変換は可逆でなければなりません。

二つの随伴の定義が等価であることを示すのは容易です。例えば、ホム集合の同型から出発して単位変換を導き出しましょう:

$$\mathbf{C}(Ld, c) \cong \mathbf{D}(d, Rc)$$

この同型が任意の対象 c に対して機能するので、それは $c = Ld$ に対しても機能しなければなりません:

$$\mathbf{C}(Ld, Ld) \cong \mathbf{D}(d, (R \circ L)d)$$

左側には少なくとも一つの射、恒等射が含まれていることがわかります。自然変換はこの射を $\mathbf{D}(d, (R \circ L)d)$ の要素、つまり恒等関手 I を挿入して:

$$\mathbf{D}(Id, (R \circ L)d)$$

の射へ写像します。 d によってパラメータ化された射の族を得ます。それらは関手 I と関手 $R \circ L$ 間の自然変換を形成します(自然性条件は容易に検証できます)。これはまさに私たちの単位 η です。

逆に、単位と余単位の存在から出発して、ホム集合間の変換を定義することができます。例えば、 $\mathbf{C}(Ld, c)$ のホム集合の任意の射 f を選びましょう。 φ を定義したいと思いますが、それは f に作用して $\mathbf{D}(d, Rc)$ の射を生成します。

実際に選択肢はありません。一つの試みとして、 f を R を使って引き上げることができます。それは $R(Ld)$ から Rc への射 Rf を生成します— $\mathbf{D}((R \circ L)d, Rc)$ の射です。

私たちが必要とする φ のコンポーネントは、 d から Rc への射です。それは問題ありません、なぜなら η_d のコンポーネントを使って d から $(R \circ L)d$ へ行くことができます。私たちは得ます:

$$\varphi_f = Rf \circ \eta_d$$

逆方向は類推的であり、 ψ の導出も同様です。

Haskell の **Adjunction** の定義に戻ると、自然変換 φ と ψ はそれぞれ **leftAdjunct** と **rightAdjunct** という **a** と **b** における多相的 (polymorphic) 関数に置き換えられます。関手 L と R は **f** と **u** と呼ばれます:

```
class (Functor f, Representable u) =>
  Adjunction f u | f -> u, u -> f where
    leftAdjunct :: (f a -> b) -> (a -> u b)
    rightAdjunct :: (a -> u b) -> (f a -> b)

abstract class Adjunction[F[_], U[_]](
  implicit val F: Functor[F],
  val U: Representable[U]
){ 
  // changed the order of parameters
  // to help Scala compiler infer types
}
```

```
def leftAdjunct[A, B](a: A)(f: F[A] => B): U[B]

def rightAdjunct[A, B](fa: F[A])(f: A => U[B]): B
}
```

`unit/counit` の定義と `leftAdjunct/rightAdjunct` の定義との間の同等性は、これらの写像によって証明されます:

```
unit          = leftAdjunct id
counit        = rightAdjunct id
leftAdjunct f = fmap f . unit
rightAdjunct f = counit . fmap f

def unit[A](a: A): U[F[A]] =
    leftAdjunct(a)(identity)

def counit[A](a: F[U[A]]): A =
    rightAdjunct(a)(identity)

def leftAdjunct[A, B](a: A)(f: F[A] => B): U[B] =
    U.map(unit(a))(f)

def rightAdjunct[A, B](a: F[A])(f: A => U[B]): B =
    counit(F.map(a)(f))
```

圈の記述から Haskell コードへの翻訳を追うことは非常に教育的です。これを練習として強く推奨します。

Haskellにおいて、右随伴が自動的に表現可能関手である理由を説明する準備ができました。これは、第一近似として、Haskellの型の圏を集合の圏として扱うことができるためです。

右圏 \mathbf{D} が \mathbf{Set} の場合、右随伴 R は \mathbf{C} から \mathbf{Set} への関手です。そのような関手が表現可能である場合、私たちは \mathbf{C} 内のある対象 rep を見つけることができ、ホム関手 $\mathbf{C}(rep, _)$ が R と自然に同型であると言えます。実際には、もし R が \mathbf{Set} から \mathbf{C} へのいくつかの関手 L の右随伴である場合、そのような対象は常に存在します—それは単集合 $()$ の下での L の像です:

$$rep = L()$$

確かに、随伴によれば、次の二つのホム集合が自然に同型です:

$$\mathbf{C}(L(), c) \cong \mathbf{Set}(((), Rc)$$

与えられた c に対して、右側は単集合 $()$ から Rc への関数の集合です。私たちは以前に見たように、各そのような関数が Rc の集合から一つの要素を選びます。その関数の集合は Rc の集合と同型です。ですから、私たちは持っています:

$$\mathbf{C}(L(), -) \cong R$$

これは R が確かに表現可能であることを示しています。

18.3 随伴からの積

私たちは普遍構成を使用していくつかの概念を以前に紹介しました。それらの概念は、大局的に定義されるとき、随伴を使用して表現する方が容易です。最も簡単な非自明な例は積の概念です。**積の普遍構成**の要点は、任意の積候補を普遍的積を通じて因数分解する能力です。

より正確には、二つの対象 a と b の積は対象 $(a \times b)$ (または Haskell の記法で `(a, b)`) であり、二つの射 fst と snd で装備されており、任意の他の候補 c が二つの射 $p :: c \rightarrow a$ と $q :: c \rightarrow b$ で装備されている場合、 fst と snd を通じて p と q を因数分解する一意の射 $m :: c \rightarrow (a, b)$ が存在します。

前に見たように、Haskell では、この射を生成する `factorizer` を実装することができます:

```
factorizer :: (c -> a) -> (c -> b) -> (c -> (a, b))
factorizer p q = \x -> (p x, q x)

def factorizer[A, B, C](p: C => A)(q: C => B): (C => (A, B)) =
  x => (p(x), q(x))
```

因数分解条件が成り立つことを検証するのは簡単です:

```
fst . factorizer p q = p
snd . factorizer p q = q
```

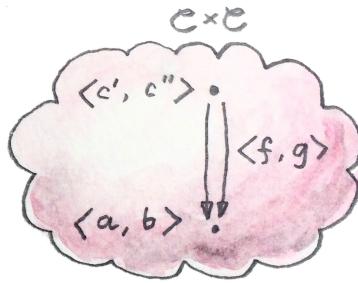
```
factorizer(p)(q).andThen(_.1) == p  
factorizer(p)(q).andThen(_.2) == q
```

私たちは、射のペア p と q を取り、別の射 $m = \text{factorizer } p \ q$ を生成する写像を持っています。

これを随伴の定義に必要な二つのホム集合間の写像にどのように翻訳するかですか？コツは **Hask** の外に出て、射のペアを单一の射として扱う積圏を考えることです。

積圏とは何かを思い出させてください。任意の二つの圏 C と D を取ります。積圏 $C \times D$ 内の対象は、 C から一つと D から一つ、二つの対象のペアです。射は、 C から一つと D から一つの射のペアです。

ある圏 C 内の積を定義するには、積圏 $C \times C$ から始めるべきです。 C からの射のペアは、積圏 $C \times C$ 内の单一の射です。



最初は少し混乱するかもしれません、積を定義するために積圏を使用していることは、これらが非常に異なる積であることを理解していく

ださい。積圏を定義するためには普遍構成が必要ありません。対象のペアと射のペアの概念が必要です。

しかし、 \mathbf{C} からの対象のペアは \mathbf{C} 内の対象ではありません。それは別の圏、 $\mathbf{C} \times \mathbf{C}$ 内の対象です。形式的には、 \mathbf{C} の対象 a と b を $\langle a, b \rangle$ として書くことができます。一方、普遍構成は、 \mathbf{C} 内の同じ圏内の対象 $a \times b$ (または Haskell での (a, b)) を定義するために必要です。この対象は、普遍構成で指定された方法で $\langle a, b \rangle$ を代表することになります。それは常に存在するわけではなく、ある対象のペアに対して存在する場合でも、 \mathbf{C} 内の他のペアに対しては存在しないかもしれません。

では、**factorizer** をホム集合の写像として見てみましょう。最初のホム集合は積圏 $\mathbf{C} \times \mathbf{C}$ 内にあり、二番目のものは \mathbf{C} 内にあります。一般的な射は $\mathbf{C} \times \mathbf{C}$ 内で射のペア $\langle f, g \rangle$ です:

$$\begin{aligned}f &:: c' \rightarrow a \\g &:: c'' \rightarrow b\end{aligned}$$

c'' は c' と異なる可能性があります。しかし、積を定義するためには、同じ出発対象 c を共有する特別な射、 p と q のペアに興味があります。それは大丈夫です: 随伴の定義では、左ホム集合の出発点は任意の対象ではありません—それは右圏からの何らかの対象に作用する左関手 L の結果です。当てはまる関手は対角関手 Δ です。それは \mathbf{C} から $\mathbf{C} \times \mathbf{C}$ への関手で、対象への作用は次のようにになります:

$$\Delta c = \langle c, c \rangle$$

したがって、私たちの随伴の左側のホム集合は次のようにになります:

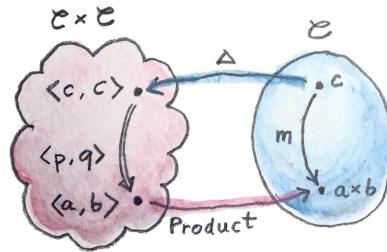
$$(\mathbf{C} \times \mathbf{C})(\Delta c, \langle a, b \rangle)$$

それは積圏のホム集合です。その要素は **factorizer** への引数として認識される射のペアです:

$$(c \rightarrow a) \rightarrow (c \rightarrow b) \dots$$

右側のホム集合は \mathbf{C} 内にあり、それは出発対象 c といくつかの関手 R が $\mathbf{C} \times \mathbf{C}$ 内の目標対象に作用した結果との間にあります。それは $\langle a, b \rangle$ のペアを私たちの積対象 $a \times b$ に写像する関手です。このホム集合の要素は **factorizer** の結果として認識されます:

$$\dots \rightarrow (c \rightarrow (a, b))$$



私たちはまだ完全な随伴を持っていません。それにはまず、私たちの **factorizer** が可逆である必要があります—私たちはホム集合間の同型を構成しています。**factorizer** の逆は、ある対象 c から積対象 $a \times b$

への射 m から始まるべきです。言い換えると、 m は以下の要素でなければなりません:

$$C(c, a \times b)$$

逆の因数分解器は m を $\langle c, c \rangle$ から $\langle a, b \rangle$ への射 $\langle p, q \rangle$ に写像する必要があります。言い換えると、それは以下の要素の射です:

$$(C \times C)(\Delta c, \langle a, b \rangle)$$

その写像が存在する場合、私たちは対角関手の右随伴が存在すると結論付けます。その関手は積を定義します。

Haskell では、`m` をそれぞれ `fst` と `snd` で合成することによって、常に `factorizer` の逆を構成することができます。

```
p = fst . m
q = snd . m
```

二つの方法で積を定義することの等価性の証明を完了するためには、ホム集合間の写像が a 、 b 、そして c で自然であることを示す必要があります。これは献身的な読者にとっての演習として残します。

要約すると、私たちが行ったこと: 圏の積は、対角関手の**右随伴**として大局的に定義されるかもしれません:

$$(C \times C)(\Delta c, \langle a, b \rangle) \cong C(c, a \times b)$$

ここで、 $a \times b$ は私たちの右随伴関手 *Product* の作用の結果です。 $\langle a, b \rangle$ のペアに作用します。任意の関手から $C \times C$ へは双関手ですので、

Product も双関手です。Haskell では、*Product* 双関手は単純に $(,)$ として書かれます。それを二つの型に適用して、例えばその積型を得ることができます:

```
(,) Int Bool ~ (Int, Bool)
```

```
Product2[Int, Boolean] ~ (Int, Boolean)
```

18.4 随伴からの指數

指數 b^a 、または関数対象 $a \Rightarrow b$ は、**普遍構成**を使用して定義することができます。この構成が全ての対象のペアに対して存在する場合、それは随伴として見ることができます。再び、コツは以下の声明に集中することです:

任意の他の対象 z と射 $g :: z \times a \rightarrow b$ がある場合、一意の射 $h :: z \rightarrow (a \Rightarrow b)$ が存在します。

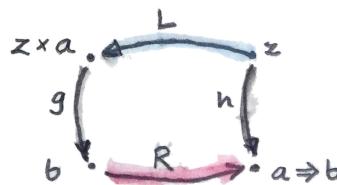
この声明はホム集合間の写像を確立します。

この場合、私たちは同じ圏内の対象に取り組んでいるので、二つの随伴関手は自己関手です。左の自己関手 L は、対象 z に作用すると $z \times a$ を生成します。それは、いくつかの固定された a との積を取る関手に対応します。

右の自己関手 R は、 b に作用すると関数対象 $a \Rightarrow b$ （または b^a ）を生成します。再び、 a は固定されています。これら二つの関手の間の随伴はしばしば次のように書かれます：

$$-\times a \dashv (-)^a$$

この随伴に基づいているホム集合の写像は、私たちが普遍構成で使用した図を再描画することによって最もよく見ることができます。



eval 射^{*1}はこの随伴の余単位に他なりません：

$$(a \Rightarrow b) \times a \rightarrow b$$

ここで：

$$(a \Rightarrow b) \times a = (L \circ R)b$$

私は以前に述べた通り、普遍構成は一意の対象を定義します、同型までです。だから私たちは「積」と「指數」を持っています。これは随伴にも翻訳されます：もし関手が随伴を持つならば、この随伴は同型までに一意です。

^{*1} 第9章の普遍構成を参照。

18.5 チャレンジ

1. ψ の自然性四角形を導き出してください。これは二つの (反変) 関手間の変換です:

$$a \rightarrow \mathbf{C}(La, b)$$

$$a \rightarrow \mathbf{D}(a, Rb)$$

2. 第二の随伴定義のホム集合の同型から余単位 ε を導き出してください。
3. 二つの随伴定義の等価性の証明を完成させてください。
4. 余積が随伴によって定義できることを示してください。余積の因数分解器の定義から始めてください。
5. 余積が対角関手の左随伴であることを示してください。
6. Haskell で積と関数対象間の随伴を定義してください。

19

自由/忘却随伴

隨伴の強力な応用としての**自由関手**は、**忘却関手**の左随伴として定義されます。忘却関手は通常、ある構造を忘れるという、かなりシンプルな関手です。例えば、多くの興味深い圏は集合の上に構成されています。しかし、これらの集合を抽象化した圏論的対象は内部構造を持たず、要素を持ちません。それでも、これらの対象はしばしば集合の記憶を保持しており、ある圏 C から Set への射としての写像 – 関手 – が存在します。ある対象の C に対応する集合は、その**台集合**と呼ばれます。

モノイドは要素の集合を台集合として持つような対象です。モノイドの圏 Mon から集合の圏への忘却関手 U は、モノイドをそれらの台

集合へと写像し、モノイドの射(準同型)を集合間の関数へと写像します。

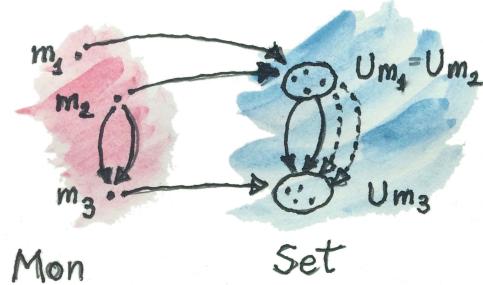
Mon は分裂した性格を持つと考えるのが好きです。一方で、それは乗法と単位要素を持つ集合の束です。一方で、それは対象が特徴のないもので、唯一の構造がその間にある射によって符号化される圏です。乗法と単位を保つすべての集合関数は、**Mon** における射を生み出します。

心に留めておくべきこと:

- 同じ集合に写像されるモノイドが多数存在するかもしれません。
- モノイドの射は、それらの台集合間の関数よりも少ない(または多くとも同じ数)です。

忘却関手 U に対する左随伴である関手 F は、生成集合から自由モノイドを構成する自由関手です。この随伴関係は、以前議論した自由モノイドの普遍構成から導かれます。^{*1}

^{*1} 第 13 章の **自由モノイド** を参照してください。



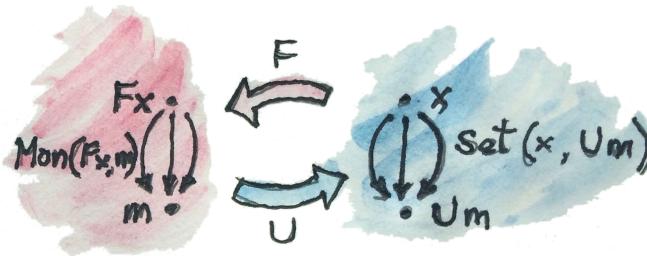
モノイド m_1 と m_2 は同じ台集合を持ちます。 m_2 と m_3 の台集合間の関数は、それらの間の射よりも多いです。

ホム集合の観点から、この随伴関係を次のように書くことができます:

$$\mathbf{Mon}(Fx, m) \cong \mathbf{Set}(x, Um)$$

この (x と m に対して自然な) 同型は、以下を示しています:

- x によって生成される自由モノイド Fx と任意のモノイド m 間のすべてのモノイド準同型に対して、生成集合 x を m の台集合に埋め込む一意的な関数が存在します。それは $\mathbf{Set}(x, Um)$ 内の関数です。
- ある m の台集合に x を埋め込むすべての関数に対して、 x によって生成された自由モノイドとモノイド m 間の一意的な射が存在します。(これは私たちが普遍構成で呼んだ射 h です。)



直感的には、 Fx は x の基礎に構成できる「最大」のモノイドです。もし私たちがモノイドの内部を見ることができたなら、 $\mathbf{Mon}(Fx, m)$ に属するどの射も、他のモノイド m にこの自由モノイドを埋め込むことを確認できるでしょう。それは、おそらくいくつかの要素を同一視することによって行われます。特に、 Fx の生成要素（つまり x の要素）を m に埋め込みます。随伴関係は、右側にある $\mathbf{Set}(x, Um)$ からの関数によって与えられる x の埋め込みが、左側にあるモノイドの埋め込みを一意に決定すること、そしてその逆もまた真であることを示しています。

Haskell では、リストデータ構造は自由モノイドです（いくつかの注意点があります: [Dan Doel's blog post^{*2}](#) を参照）。リスト型 `[a]` は、生成集合を表す型 `a` を持つ自由モノイドです。例えば、型 `[Char]` は、単位要素 – 空のリスト `[]` – および単集合 `['a'], ['b']` – 自由モノイドの生成要素を含みます。残りは「積」を適用することによって生成されます。ここでの積は、単に一方のリストをもう一方に追加すること

^{*2} <http://comonad.com/reader/2015/free-monoids-in-haskell/>

とです。追加は結合的で単位的です(つまり、中立要素、ここでは空のリストがあります)。型 `Char` によって生成される自由モノイドは、`Char` からのすべての文字列の集合に他なりません。これは Haskell では `String` と呼ばれます:

```
type String = [Char]
```

```
type String = List[Char]
```

(`type` は既存の型の別名、すなわち型同義名を定義します)。

もう一つの興味深い例は、ただ一つの生成要素から構成される自由モノイドです。それは単位のリストの型、`[()` です。その要素は `[]`, `[()]`, `[(), ()]`, などです。そのようなリストの各々は、その長さという一つの自然数によって記述することができます。単位のリストにはそれ以上の情報はエンコードされていません。二つのそのようなリストを追加すると、その構成要素の長さの和である新しいリストが生成されます。型 `[()` が、自然数(ゼロを含む)の加法モノイドと同型であることは容易にわかります。ここに、互いに逆である二つの関数があり、この同型性を証明しています:

```
toNat :: [()] -> Int
toNat = length

toLst :: Int -> [()]
toLst n = replicate n ()
```

```
def toNat: List[Unit] => Int =  
  _.length  
  
def toLst: Int => List[Unit] =  
  n => List.fill(n)(())
```

単純化のために `Int` 型を使用しましたが、考え方は同じです。`replicate` 関数は、指定された値 (ここでは単位) で事前に満たされた長さ `n` のリストを作成します。

19.1 いくつかの直感

以下に述べるのは、いくつかの手振り的な議論です。これらの種類の議論は厳密からは程遠いですが、直感を形成するのに役立ちます。

自由/忘却随伴についての直感を得るために、関手と関数が本質的に情報損失を伴うということを心に留めておくと役立ちます。関手は複数の対象と射を折りたたむことがあります、関数は集合の複数の要素をまとめることができます。また、それらの像は終域の一部のみをカバーするかもしれません。

`Set` 内の「平均的な」ホム集合には、最も情報損失が少ないもの (例えば、単射や、可能であれば同型) から、始域全体を单一の要素に収束させる定数関数まで、全ての関数のスペクトルが含まれるでしょう。

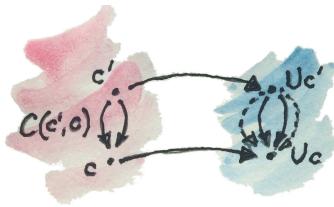
私は、任意の圏における射もまた情報損失を伴うものとして考えることがあります。これは単なる精神モデルですが、随伴関係を考えるとき、特に一方の圏が **Set** である場合に、有用なモデルです。

形式的には、射が可逆である(同型)かそうでないかのみを議論することができます。後者の種類の射は、情報損失を伴うものと考えられるかもしれません。また、单射(非収束)や全射(全終域をカバー)関数を一般化するモノ射やエピ射の概念もありますが、モノでありエピでありながら、それでも可逆でない射が存在する可能性があります。

Free \boxtimes Forgetful 随伴では、より制約された圏 **C** を左に、そしてより少ない制約を持つ圏 **D** を右に持ります。**C** 内の射は「少ない」です。なぜなら、それらは追加の構造を保持する必要があるからです。**Mon** の場合、それらは乗法と単位を保持する必要があります。**D** 内の射は、そこまで多くの構造を保持する必要がないため、「より多い」です。

ある対象 c を **C** から忘却関手 U を使って写像するとき、私たちはそれを c の「内部構造」を明らかしていると考えます。実際、**D** が **Set** である場合、私たちは U が c の内部構造—その台集合—を定義していると考えます。(任意の圏では、他の対象との関連を通してしか対象の内部について話すことができませんが、ここでは手を振っているだけです。)

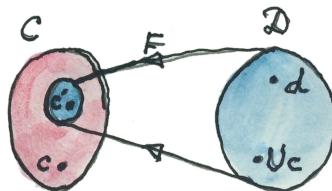
二つの対象 c' と c を U で写像するとき、一般的には $C(c', c)$ のホム集合の写像が $D(Uc', Uc)$ の部分集合のみをカバーすると期待されます。これは、 $C(c', c)$ の射が追加の構造を保持する必要がある一方で、 $D(Uc', Uc)$ の射はそうする必要がないためです。



しかし、随伴関係が特定のホム集合の**同型**として定義されているので、 c' の選択に非常に注意しなければなりません。随伴関係では、 c' は C のどこからでも選ばれるのではなく、自由関手 F の（おそらく小さい）像から選ばれます：

$$C(Fd, c) \cong D(d, Uc)$$

したがって、 F の像は、任意の c に多くの射を持つ対象から構成されなければなりません。実際には、 Fd から c への構造を保持する射が、 d から Uc への構造を保持しない射と同じ数だけ存在する必要があります。これは、 F の像が本質的に構造のない「自由」対象でなければならないことを意味します。このような「構造のない」対象は自由対象と呼ばれます。



モノイドの例では、自由モノイドは単位と結合則によって生成される以外の構造を持ちません。それ以外のすべての乗法は新しい要素を生み出します。

自由モノイドでは、 $2 * 3$ は 6 ではなく新しい要素 $[2, 3]$ です。 $[2, 3]$ と 6 を同一視しないため、この自由モノイドから任意の他のモノイド m への射は、それらを別々に写像することも、両方を m の同じ要素に写像することも、あるいは $[2, 3]$ と 5 (それらの和) を加法モノイドで同一視することも許されます。異なる同一視が異なるモノイドを生み出します。

これは別の興味深い直感につながります: 自由モノイドは、モノイド演算を実行する代わりに、それに渡された引数を蓄積します。2 と 3 を掛ける代わりに、2 と 3 をリストに覚えています。このスキームの利点は、どのモノイド演算を使用するかを指定する必要がないことです。引数を蓄積し続け、最後に結果に演算子を適用することができます。そしてその時に初めて、どの演算子を適用するかを選ぶことができます。数字を足すことも、掛けることも、2 を法とする加算を行うこともできます。自由モノイドは、式の作成と評価を分離します。代数について話すときに、この考え方を再び見ることになります。

この直感は、他により複雑な自由構成にも一般化することができます。たとえば、評価する前に全体の式木を蓄積することができます。このアプローチの利点は、評価を高速化したり、メモリ消費を少なくするために、そのような木を変換できることです。これは、例えば、

行列計算を実装する際に行われます。そこでは、積極的な評価が中間結果を格納する一時的な配列の多大な割り当てをもたらすでしょう。

19.2 チャレンジ

1. 単一の生成要素から構成される自由モノイドを考えてください。この自由モノイドから任意のモノイド m への射と、单一の生成要素の集合から m の台集合への関数との間に一一対応があることを示してください。

20

モナド: プログラマによる定義

プログラマはモナドについて様々な神話を作り上げています。それはプログラミングにおいて最も抽象的で難解な概念の一つとされています。それを「理解する」人とそうでない人がいます。多くの人にとって、モナドの概念を理解する瞬間は神秘的な体験のようなものです。モナドは多岐にわたる構造の本質を抽象化しているため、日常生活においてそれに相当する良い類似物がありません。私たちは暗闇の中を手探りで進むようなもので、盲目の人が象の異なる部分を触りながら勝ち誇って「これは綱だ」「これは木の幹だ」あるいは「これはブリトーだ」と叫ぶようなものです。

ここで真実を明らかにしましょう: モナドを取り巻く全ての神秘性は誤解から生まれています。モナドは非常にシンプルな概念です。混乱を招くのはモナドの応用の多様性です。

この投稿のための研究の一環として、ダクトテープ(いわゆるダックテープ)とその応用を調べました。これがいくつかの例です:

- ダクトの封印
- アポロ 13 号の CO₂ スクラバーの修理
- イボ治療
- Apple の iPhone 4 の通話落ち問題の修正
- プロムドレスの製作
- 吊り橋の建設

さて、ダクトテープが何かを知らないとして、このリストを基にそれを理解しようとしてください。幸運を祈ります！

そこで、私が追加したいのは「モナドは...のようなもの」という陳腐な比喩のコレクションにもう一つアイテムを加えることです: モナドはダクトテープのようなものです。その応用は非常に多様ですが、その原理は非常にシンプルです: それは物事を結びつけます。より正確には、物事を合成します。

これは、特に手続き型の背景から来たプログラマがモナドを理解するのに苦労する理由を部分的に説明しています。問題は、私たちが関数合成という観点でプログラミングを考えることに慣れていないことです。これは理解しやすいことです。私たちはしばしば中間値に名前

を付けるか、それらを直接関数から関数へと渡すのではなく、短い接着コードのセグメントをインライン化するか、補助関数に抽象化することを避けます。ここに C でのベクトル長関数の手続き型スタイルの実装があります:

```
double vlen(double * v) {
    double d = 0.0;
    int n;
    for (n = 0; n < 3; ++n)
        d += v[n] * v[n];
    return sqrt(d);
}
```

これを関数合成を明示する (スタイルリッシュな) Haskell のバージョンと比較してみてください:

```
vlen = sqrt . sum . fmap (flip (^) 2)

def fmap[A, B]: (A => B) => List[A] => List[B] =
  f => {
    case Nil => Nil
    case x :: xs => f(x) :: fmap[A, B](f)(xs)
  }

def sum: List[Double] => Double = _.sum

import math._
```

```
def vlen: List[Double] => Double =  
    sqrt _ compose sum compose fmap(pow(_, 2))
```

(ここで、より神秘的にするために、私は累乗演算子 (\wedge) をその第二引数に 2 を設定して部分適用しました。)

私は Haskell のポイントフリースタイルが常に優れていると主張しているわけではありませんが、プログラミングにおいて私たちが行う全てのことの底には関数合成があります。そして、実際には関数を合成しているにも関わらず、Haskell はモナド的な合成のための手続き型スタイルの構文である `do` 表記を提供しています。後ほどその使用方法を見ていきますが、まず、なぜそもそもモナド的な合成が必要なのかを説明しましょう。

20.1 Kleisli 圈

以前、我々は通常の関数に装飾を施すことによって `Writer` モナドに到達しました。特にその装飾は、それらの戻り値を文字列とペアにすること、あるいは一般にモノイドの要素とペアにすることによって行われました。我々は今、そのような装飾が関手であることを認識できます:

```
newtype Writer w a = Writer (a, w)  
  
instance Functor (Writer w) where
```

```
fmap f (Writer (a, w)) = Writer (f a, w)

case class Writer[W, A](run: (A, W))

implicit def writerFunctor[W] = new Functor[Writer[W, ?]] {
  def fmap[A, B](f: A => B)(fa: Writer[W, A]): Writer[W, B] =
    fa match {
      case Writer((a, w)) => Writer(f(a), w)
    }
}
```

その後、我々は装飾された関数、または Kleisli 射の合成方法を見つきました。それらは形式としては以下ののような関数です:

```
a -> Writer w b  
A => Writer[W, B]
```

ログの蓄積は合成の内部で実装されました。

我々は今、Kleisli 圈のより一般的な定義に備えています。我々は圏 C と自己関手 m から始めます。対応する Kleisli 圈 K は C と同じ対象を持ちますが、その射は異なります。 K 内の二つの対象 a と b の間の射は、元の圏 C 内での射として実装されます:

$$a \rightarrow m b$$

a と b の間の \mathbf{K} 内の Kleisli 射を、 a と $m b$ の間ではなく射として扱うことが重要です。

私たちの例では、 m は `Writer w` に特化されましたが、ある固定されたモノイド w のためです。

Kleisli 射は、それらに適切な合成を定義できる場合に限り、圏を形成します。合成が存在し、それが結合的であり、すべての対象に対する恒等射がある場合、関手 m はモナドと呼ばれ、結果として得られる圏は Kleisli 圏と呼ばれます。

Haskell では、Kleisli 合成は魚演算子`>=>`を使用して定義され、恒等射は `return` と呼ばれる多相的関数です。ここにモナドの定義があります:

```
class Monad m where
    (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
    return :: a -> m a

trait Monad[M[_]] {
    def >=>[A, B, C](m1: A => M[B], m2: B => M[C]): A => M[C]
    def pure[A](a: A): M[A]
}
```

多くの同等のモナドの定義が存在し、これが Haskell のエコシステムでの主要なものではないことを念頭に置いてください。私はその概念的なシンプルさと直感的理解のためにこれを好みますが、プログラミ

ングする際には他の定義の方が便利なこともあります。間もなくそれについて話し合いましょう。

この定式化では、モナド則は非常に簡単に表現できます。それらは Haskell で強制されることはできませんが、等式推論のために使用することができます。それらは Kleisli 圏のための標準的な合成則にすぎません:

```
(f >=> g) >=> h = f >=> (g >=> h) -- 結合則  
return >=> f = f                         -- 左単位則  
f >=> return = f                          -- 右単位則
```

このような定義はまた、モナドが本当に何であるかを表現します: それは装飾された関数を合成する方法です。それは副作用や状態についてのものではありません。それは合成についてのものです。後ほど見るように、装飾された関数は様々な作用や状態を表現するために使用することができますが、それがモナドのためのものではありません。モナドは装飾された関数の一方の端を他方の端にくっつける粘着性のあるダクトテープです。

私たちの `Writer` 例に戻ると、ログ関数 (`Writer` 関手の Kleisli 射) は、`Writer` がモナドであるために圏を形成します:

```
instance Monoid w => Monad (Writer w) where  
    f >=> g = \a ->  
        let Writer (b, s) = f a  
            Writer (c, s') = g b  
        in Writer (c, s' ++ s)
```

```

        in Writer (c, s `mappend` s')
    return a = Writer (a, mempty)

implicit def writerMonad[W: Monoid] = new Functor[Writer[W, ?]] {
    def >=>[A, B, C](f: A => Writer[W, B], g: B => Writer[W, C]) =
        a => {
            val Writer((b, s1)) = f(a)
            val Writer((c, s2)) = g(b)
            Writer((c, Monoid[W].combine(s1, s2)))
        }
}

def pure[A](a: A) =
    Writer(a, Monoid[W].empty)
}

object kleisliSyntax {
    //allows us to use >=> as an infix operator
    implicit class MonadOps[M[_], A, B](m1: A => M[B]) {
        def >=>[C](m2: B => M[C])(implicit m: Monad[M]): A => M[C] = {
            m.>=>(m1, m2)
        }
    }
}
}

```

モナド則は `w` のモノイド則が満たされる限り `Writer w` に対して満たされます (それらも Haskell で強制されることはありません)。

`Writer` モナドのために定義された便利な Kleisli 射があります。それは `tell` と呼ばれ、その唯一の目的はその引数をログに追加することです:

```
tell :: w -> Writer w ()  
tell s = Writer ((), s)  
  
def tell[W](s: W): Writer[W, Unit] =  
  Writer((), s)
```

後ほど他のモナド的な関数の構成ブロックとしてそれを使用します。

20.2 魚の解剖学

異なるモナドに対する魚演算子の実装を行うとすぐに、多くのコードが繰り返されることが分かり、簡単にまとめることができます。始めに、二つの関数の Kleisli 合成は関数を返す必要がありますので、その実装は型 `a` の引数を取るラムダで始まるかもしれません:

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)  
f >=> g = \a -> ...  
  
def >=>[A, B, C](f: A => M[B], g: B => M[C]) =  
  a => {...}
```

この引数でできる唯一のことは、それを `f` に渡すことです:

```
f >=> g = \a -> let mb = f a
in ...

def >=>[A, B, C](f: A => M[B], g: B => M[C]) =
a => {
val mb = f(a)
...
}
```

この時点で、私たちは型 $m\ c$ の結果を生成する必要がありますが、型 $m\ b$ のオブジェクトと関数 $g :: b \rightarrow m\ c$ が手元にあります。それを行うための関数を定義しましょう。この関数は *bind* と呼ばれ、通常は中置演算子の形で書かれます:

```
(>>=) :: m a -> (a -> m b) -> m b

def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B]
```

各モナドに対して魚演算子を定義する代わりに、bind を定義することもできます。実際、標準的な Haskell のモナドの定義は bind を使用しています:

```
class Monad m where
(>>=) :: m a -> (a -> m b) -> m b
return :: a -> m a
```

```
trait Monad[M[_]] {  
    def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B]  
    def pure[A](a: A): M[A]  
}
```

こちらは `Writer` モナドのための bind の定義です:

```
(Writer(a, w)) >>= f = let Writer(b, w') = f a  
                           in Writer(b, w `mappend` w')  
  
object bindSyntax {  
    //allows us to use flatMap as an infix operator  
    implicit class MonadOps[A, B, W: Monoid](wa: Writer[W, A]) {  
        def flatMap(f: A => Writer[W, B]): Writer[W, B] = wa match {  
            case Writer((a, w1)) =>  
                val Writer((b, w2)) = f(a)  
                Writer(b, Monoid[W].combine(w1, w2))  
        }  
    }  
}
```

確かに魚演算子の定義よりも短いですね。

さらに、`m` が関手であるという事実を利用して bind をさらに解体することができます。関数 `a -> m b` を `m a` の内容に適用するために `fmap` を使用できます。これにより、`a` は `m b` に変わります。したがって、適用の結果は型 `m (m b)` になります。これはまさに我々が求める

結果型 `m` `b` ではありませんが、近いです。必要なのは、`m` の二重適用を崩壊させるか平坦化する関数です。そのような関数は `join` と呼ばれれます:

```
join :: m (m a) -> m a

def flatten[A](mma: M[M[A]]): M[A]
```

`join` を使って、`bind` を次のように書き換えることができます:

```
ma >>= f = join (fmap f ma)

def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B] =
  flatten(fmap(f)(ma))
```

これによりモナドを定義する第三の方法がもたらされます:

```
class Functor m => Monad m where
  join :: m (m a) -> m a
  return :: a -> m a

trait Monad[M[_]] extends Functor[M] {
  def flatten[A](mma: M[M[A]]): M[A]
  def pure[A](a: A): M[A]
}
```

ここでは、`m` が **Functor** であることを明示的に要求しています。先の二つのモナドの定義ではそれを要求する必要はありませんでした。なぜなら、魚演算子または `bind` 演算子をサポートする任意の型コンストラクタ `m` は自動的に関手であるからです。例えば、`bind` と `return` を使って `fmap` を定義することが可能です:

```
fmap f ma = ma >=> \a -> return (f a)
```

```
def fmap[A, B](f: A => B)(ma: M[A]): M[B] =
  flatMap(ma)(a => pure(f(a)))
```

完全性のために、`Writer` モナドのための `join` はこちらです:

```
join :: Monoid w => Writer w (Writer w a) -> Writer w a
join (Writer ((Writer (a, w')), w)) = Writer (a, w `mappend` w')
```

```
def flatten[A, W: Monoid](wwa: Writer[W, Writer[W, A]]): Writer[W, A] =
  wwa match {
    case Writer((Writer((a, w2)), w1)) =>
      Writer(a, Monoid[W].combine(w1, w2))
  }
```

20.3 do 記法

モナドを使ってコードを書く一つの方法は、Kleisli 射を扱い、それらを魚演算子を使用して合成することです。このプログラミングスタ

イルは、ポイントフリースタイルの一般化です。ポイントフリーコードはコンパクトで、しばしば非常にエレガントです。しかし一般的には理解が難しく、暗号のようになります。そのため、ほとんどのプログラマは関数引数と中間値に名前を付けることを好みます。

モナドを扱う場合、これは魚演算子よりも bind 演算子を好むことを意味します。bind はモナド的な値を取り、モナド的な値を返します。プログラマはそれらの値に名前を付けるを選択できます。しかし、それはほとんど改善とは言えません。我々が本当に欲しいのは、モナド的コンテナに封じ込められているものではなく、通常の値として扱われるふりをすることです。それが手続き型コードの動作方法です—副作用、たとえばグローバルログの更新などはほとんどが見えないものです。そしてそれは **do** 記法が Haskell で模倣するものです。

それでは、なぜ全ての副作用を見えなくするのであればモナドを全く使わないのでしょうか？なぜ手続き型言語に固執しないのでしょうか？答えは、モナドが副作用をより良く制御することを可能にするからです。例えば、**Writer** モナドのログは関数から関数へと渡され、決してグローバルに露出しません。ログを混乱させたり、データレスを生じさせる可能性はありません。また、モナド的コードは明確に区切られ、プログラムの残りの部分から隔離されています。

do 記法はモナド的な合成のための単なるシンタックスシュガーです。表面上は手続き型コードのように見えますが、直接 bind とラムダ式の連続に翻訳されます。

例えば、以前に `Writer` モナドの Kleisli 射の合成を説明するために使用した例を取り上げます。現在の定義を使用すると、これは次のように書き換えられます:

```
process :: String -> Writer String [String]
process = upCase >=> toWords

def toWords: String -> Writer[String, List[String]] =
  s => Writer(s.split("\s+").toList, "toWords ")

def process: String -> Writer[String, List[String]] = {
  import kleisliSyntax.-
  // -Ypartial-unification should be on
  upCase >=> toWords
}
```

この関数は入力文字列の全ての文字を大文字に変換し、単語に分割し、その一方で行動のログを生成します。

`do` 記法では次のようにになります:

```
process s = do
  upStr <- upCase s
  toWords upStr

def process: String -> Writer[String, List[String]] =
  s => {
```

```
import bindSyntax._

for {
    upStr <- upCase(s)
    words <- toWords(upStr)
} yield words

}
```

ここで、`upStr` は単なる `String` ですが、`upCase` は `Writer` を生成します:

```
upCase :: String -> Writer String String
upCase s = Writer (map toUpper s, "upCase ")

def upCase: String => Writer[String, String] =
  s => Writer(s.toUpperCase, "upCase ")
```

これはコンパイラによって次のようにデシュガーリングされる `do` ブロックです:

```
process s =
  upCase s >>= \upStr ->
    toWords upStr

def process: String => Writer[String, List[String]] =
  s => {
    import bindSyntax._

    upCase(s) >>= (upStr => toWords(upStr))
  }
```

`upCase` のモナド的な結果は `String` を取るラムダに束縛されます。この文字列の名前が `do` ブロックに現れます。次の行を読むとき:

```
upStr <- upCase s
```

```
upStr <- upCase(s)
```

私たちは `upStr` が `upCase s` の結果を「得る」と言います。

`tell` をインライン化すると、擬似手続き型スタイルはさらに顕著になります。`toWords` を "toWords" という文字列をログに記録する `tell` の呼び出しで置き換え、`upStr` を分割する `words` の結果を `return` で返す呼び出しに続けます。`words` は文字列を扱う通常の関数です。

```
process s = do
    upStr <- upCase s
    tell "toWords"
    return (words upStr)
```

```
def words: String => List[String] =
  _.split("\\s+").toList
```

```
def process: String => Writer[String, List[String]] =
  s => {
    import bindSyntax.-
    for {
      upStr <- upCase(s)
```

```
_ <- tell("toWords ")
} yield words(upStr)
}
```

ここで、`do` ブロック内の各行は、デシュガーされたコード内の新しいネストされた bind を導入します:

```
process s =
  upCase s >=> upStr ->
    tell "toWords " >=> () ->
      return (words upStr)

def process: String => Writer[String, List[String]] =
  s => {
    upCase(s).flatMap { upStr =>
      tell("toWords ").flatMap { _ =>
        writerMonad.pure(words(upStr))
      }
    }
  }
```

`tell` は Unit 値を生成するので、それを次のラムダに渡す必要はありません。モナド的な結果の内容を無視する（しかしその効果ではない—ここではログへの貢献）ことはかなり一般的なので、その場合に bind を置き換える特別な演算子があります:

```
(>>) :: m a -> m b -> m b  
m >> k = m >>= (\_ -> k)
```

```
object moreSyntax {  
    // allows us to use >> as an infix operator  
    implicit class MoreOps[A, B, W: Monoid](m: Writer[W, A])  
        extends bindSyntax.MonadOps[A, B, W](m) {  
            def >>(k: Writer[W, B]): Writer[W, B] =  
                m >>= (_ => k)  
        }  
}
```

私たちのコードの実際のデシュガーは次のようにになります:

```
process s =  
    upCase s >>= \upStr ->  
        tell "toWords " >>  
        return (words upStr)  
  
def process: String => Writer[String, List[String]] = s => {  
    import moreSyntax._  
    upCase(s) flatMap (upStr =>  
        tell("toWords ") >>  
        writerMonad.pure(words(upStr)))  
}
```

一般的に、**do** ブロックはコードの残りの部分で利用可能になる新しい名前を導入する左射を使用する行(またはサブブロック)、または純粋に副作用のために実行される行で構成されます。行の間には暗黙の bind 演算子があります。ちなみに、Haskell では **do** ブロックの書式をブレースとセミコロンに置き換えることが可能です。これはモナドをセミコロンをオーバーロードする方法として説明する正当化を提供します。

do 記法のデシュガーにおけるラムダと bind 演算子のネスティングは、各行の結果に基づいて **do** ブロックの残りの実行を影響させる効果を持ちます。この性質は、例えば例外をシミュレートするためなど、複雑な制御構造を導入するために使用することができます。

興味深いことに、**do** 記法の同等物は、特に C++ で、手続き型言語に適用されています。私が言及しているのは再開可能関数やコルーチンです。C++ の **futures** がモナドを形成する^{*1}ことは秘密ではありません。それは継続モナドの一例であり、私たちはまもなくそれについて議論します。継続は非常に合成しにくいという問題があります。Haskell では **do** 記法を使用して「私のハンドラーがあなたのハンドラーを呼び出します」のスペティを、非常に手続き型のようなものに変換します。再開可能関数は C++ で同じ変換を可能にします。そして同じメカニズムは、入れ子になったループのスペティ^{*2}をリスト

^{*1} <https://bartoszmilewski.com/2014/02/26/c17-i-see-a-monad-in-your-future/>

^{*2} <https://bartoszmilewski.com/2014/04/21/getting-lazy-with-c/>

内包表記または「ジェネレーター」に変換するために適用されることがあります、それらは本質的にリストモナドのための `do` 記法です。モナドの一元化された抽象化がなければ、これらの問題は通常、言語へのカスタム拡張によって個別に対処されます。Haskell では、これはすべてライブラリを通じて処理されます。

21

モナドと効果

モナドが何のためにあるかを理解した今 – それは飾り付けられた関数を合成するため – 本当に興味深い問題は、なぜそのような関数が関数型プログラミングで重要なのかです。既に `Writer` モナドの例を見ましたが、飾り付けによって複数の関数呼び出しを通じてログを作成・蓄積することができました。これは通常、非純粋関数(例えば、グローバル状態へのアクセスや変更)を使って解決される問題でしたが、純粋関数を用いて解決されました。

21.1 問題点

以下は、伝統的に関数の純粹性を放棄することで解決されてきた類似の問題の短いリストです。これは Eugenio Moggi の画期的な論文^{*1}からの引用です。

- 部分性: 終了しない可能性がある計算
- 非決定性: 多くの結果を返す可能性がある計算
- 副作用: 状態にアクセス／変更する計算
 - 読み取り専用状態、または環境
 - 書き込み専用状態、またはログ
 - 読み書き状態
- 例外: 失敗する可能性がある部分関数
- 繙続: プログラムの状態を保存し、要求に応じてそれを復元する能力
- 対話型入力
- 対話型出力

これらの問題が同じ巧妙な技、すなわち飾り付けられた関数に頼ることで解決されるというのは本当に驚くべきことです。もちろん、各ケースでの飾り付けは全く異なります。

^{*1} <https://core.ac.uk/download/pdf/21173011.pdf>

この段階で、飾り付けがモナド的である必要はありません。それは、私たちが合成を主張する – 単一の飾り付けられた関数をより小さな飾り付けられた関数に分解することができる – 時にのみ必要になります。再び、各装飾が異なるため、モナド的な合成は異なる方法で実装されますが、全体的なパターンは同じです。それは、恒等性を備えた結合的な合成の非常に単純なパターンです。

次のセクションは Haskell の例に重点を置いています。もしあなたが圈論に戻ることを切望しているか、または Haskell のモナドの実装に既に精通しているならば、適宜読み飛ばすか完全にスキップしてください。

21.2 解決策

まず、`Writer` モナドを使用した方法を分析しましょう。私たちは特定のタスクを実行する純粋関数から始めました – 引数が与えられたら、それは特定の出力を生成します。私たちはこの関数を、元の出力と文字列をペアにすることで飾り付けた別の関数に置き換えました。これがログ問題の解決策でした。

しかし、一般的には、私たちは单一の解決策で満足することはできません。一つのログ生成関数をより小さなログ生成関数に分解できるようにする必要がありました。これらの小さな関数の合成が私たちをモナドの概念へと導きました。

飾り付けられた関数の戻り値型を変更するというのは、通常純粹性を放棄する必要がある多くの問題に対する効果的な解決策であることが驚くべきです。リストを通して、それぞれの問題に適用される飾り付けを特定してみましょう。

21.2.1 部分性

終了しない可能性があるすべての関数の戻り値型を変更し、「持ち上げられた」型 – 元の型のすべての値に特別な「ボトム」値 ⊥ を加えた型にします。例えば、**Bool** 型は、集合として二つの要素: **True** と **False** を含みます。持ち上げられた **Bool** には三つの要素が含まれます。持ち上げられた **Bool** を返す関数は **True** または **False** を生成するか、永遠に実行されるかもしれません。

面白いことに、Haskell のような怠惰な言語では、決して終わらない関数が実際には値を返すことがあります、この値は次の関数に渡されることがあります。私たちはこの特別な値をボトムと呼びます。この値が明示的に必要でない限り(例えば、パターンマッチングするため、または出力として生成するため)、それはプログラムの実行を停止させることなく渡されるかもしれません。Haskell のすべての関数が潜在的に非終了であるため、Haskell のすべての型は持ち上げられたものと見なされます。これが、私たちがしばしば Haskell(持ち上げられた)型と関数の圈 **Hask**について話す理由ですが、**Hask** が実際には本当

の圈であるかどうかははっきりしていません（この Andrej Bauer の投稿^{*2}を参照）。

21.2.2 非決定性

多くの異なる結果を返す可能性がある関数は、一度にそれらをすべて返すこともできます。意味的には、非決定的な関数は結果のリストを返す関数と同等です。これは怠惰なガベージコレクション言語で非常に理にかなっています。例えば、1つの値だけが必要な場合、リストの先頭を取り出すことができますし、リストの後尾は評価されることはありません。ランダムな値が必要な場合は、乱数生成器を使用してリストの n 番目の要素を選ぶことができます。怠惰さは無限の結果のリストを返すことさえ可能にします。

リストモナド – Haskell における非決定的計算の実装 – で `join` は `concat` として実装されます。覚えておいてください、`join` はコンテナのコンテナを平らにすることを意図しています – `concat` はリストのリストを 1 つのリストに連結します。そして `return` は单一要素のリストを作成します:

```
instance Monad [] where
    join = concat
    return x = [x]
```

^{*2} <http://math.andrej.com/2016/08/06/hask-is-not-a-category/>

```
implicit val listMonad = new Monad[List] {  
    def flatten[A](mma: List[List[A]]): List[A] =  
        mma.reduce(_ ++ _)  
  
    def pure[A](a: A): List[A] = List(a)  
}
```

リストモナドのための bind 演算子は、一般的な公式によって与えられます: `fmap` の後に `join` が続きます。この場合は以下のようにになります:

```
as >>= k = concat (fmap k as)  
  
def flatMap[A, B](as: List[A])(k: A => List[B]): List[B] =  
    flatten(fmap(k)(as))
```

ここで、関数 `k` 自体がリストを生成するものとして、リスト `as` の各要素に適用されます。結果はリストのリストであり、それは `concat` を使って平らにされます。

プログラマの視点からは、例えば非決定的な関数をループで呼び出したり、反復子を返す関数を実装するよりも、リストを扱う方が簡単です。

ゲームプログラミングにおいて非決定性を創造的に使用するのは良い例です。例えば、コンピュータが人間に對してチェスをする場合、相手の次の動きを予測することはできません。しかし、それはすべて

の可能な動きのリストを生成し、それらを 1 つずつ分析することができます。同様に、非決定的なパーサーは与えられた式のすべての可能な解析のリストを生成することができます。

関数がリストを返すことは非決定的として解釈することができますが、リストモナドの適用ははるかに広範です。それは、命令型プログラミングで使われる反復的な構造 – ループ – の完璧な関数型代替です。1 つのループはしばしば、リストの各要素にループの本体を適用する `fmap` を使用して書き直すことができます。リストモナドにおける `do` 表記は、複雑な入れ子のループを置き換えるために使用することができます。

私のお気に入りの例は、ピタゴラスの三つ組 – 正しい三角形の辺を形成することができる正の整数の三つ組 – を生成するプログラムです。

```
triples = do
  z <- [1..]
  x <- [1..z]
  y <- [x..z]
  guard (x^2 + y^2 == z^2)
  return (x, y, z)
```

```
// Streams in Scala can be
// thought of as lazy lists
def triples = for {
  z <- Stream.from(1)
```

```
x <- 1 to z
y <- x to z
_ <- guard(x * x + y * y == z * z)
} yield (x, y, z)
```

最初の行は、`z` が無限の正の数のリスト `[1..]` から要素を取ることを教えています。その後、`x` は(有限の)リスト `[1..z]` から要素を取ります。このリストには 1 から `z` までの数が含まれています。最後に、`y` は `x` と `z` の間の数値のリストから要素を取ります。この時点で、 $1 \leq x \leq y \leq z$ の 3 つの数を使用できます。関数 `guard` は `Bool` 式を取り、`Unit` のリストを返します:

```
guard :: Bool -> [Unit]
guard True = [Unit]
guard False = []
```

```
def guard: Boolean => List[Unit] = {
  case true => List(())
  case false => List.empty[Unit]
}
```

この関数(より大きなクラスである `MonadPlus` のメンバー)は、ここでは非ピタゴラス三つ組をフィルタリングするために使用されています。実際、bind(または関連する演算子 `>>`)の実装を見ると、空のリストが与えられた場合、それは空のリストを生成することがわかります。一方、空でないリスト(ここでは、`Unit` を含む单一要素のリスト

`[()])` が与えられた場合、`bind` は続行を呼び出し、ここでは `return (x, y, z)` で、検証済みのピタゴラス三つ組を含む单一要素のリストを生成します。これらの单一要素のリストは、結果的な(無限の)結果を生成するために、囲んでいる `bind` によって連結されます。もちろん、`triples` の呼び出し側は決してリスト全体を消費することはできませんが、それは問題ではありません、なぜなら Haskell は怠惰だからです。

通常、3つのネストしたループを必要とする問題が、リストモナドと `do` 表記の助けを借りて劇的に簡素化されました。それだけでなく、Haskell はこのコードをさらに簡素化することを可能にしています。それはリスト内包表記を使用することです:

```
triples = [(x, y, z) | z <- [1..]
                      , x <- [1..z]
                      , y <- [x..z]
                      , x^2 + y^2 == z^2]
```

```
def triples = for {
    z <- Stream.from(1)
    x <- 1 to z
    y <- x to z
    if x * x + y * y == z * z
} yield (x, y, z)
```

これはリストモナドのためのさらなるシンタックスシュガーに過ぎません（厳密に言うと、`MonadPlus`）。

他の関数型または命令型言語でジェネレーターやコレーチンの様相の下で類似の構造を見るかもしれません。

21.2.3 読み取り専用状態

何らかの外部状態、または環境に読み取り専用アクセスを持つ関数は、その環境を追加引数として取る関数に常に置き換えることができます。純粋関数 $(a, e) \rightarrow b$ (ここで e は環境の型) は、最初の見た目には Kleisli 射のようではありません。しかし、私たちがそれを $a \rightarrow (e \rightarrow b)$ に Curry 化するとすぐに、飾り付けを私たちの古い友人である Reader 関手として認識します：

```
newtype Reader e a = Reader (e -> a)
```

```
case class Reader[E, A](run: E => A)
```

`Reader` を返す関数を、環境を与えられた結果を生成するアクション、すなわちミニ実行可能ファイルとして解釈することができます。そのようなアクションを実行するためのヘルパー関数 `runReader` があります：

```
runReader :: Reader e a -> e -> a
runReader (Reader f) e = f e
```

```
def runReader[E, A]: Reader[E, A] => E => A = {
  case Reader(f) => e => f(e)
}
```

それは環境の異なる値に対して異なる結果を生成するかもしれません。

`Reader` を返す関数も、`Reader` アクション自体も純粋です。

`Reader` モナドのための bind を実装するために、まず環境 `e` を取り、`b` を生成する関数を作成する必要があります:

```
ra >>= k = Reader (\e -> ...)
```

```
def flatMap[A, B](ra: Reader[E, A])(k: A => Reader[E, B]): Reader[E, B] =
  Reader { e => ... }
```

ラムダの中で、私たちはアクション `ra` を実行して `a` を生成することができます:

```
ra >>= k = Reader (\e -> let a = runReader ra e
                      in ...)
```

```
def flatMap[A, B](ra: Reader[E, A])(k: A => Reader[E, B]): Reader[E, B] =
  Reader { e =>
    val a = runReader(ra)(e)
    ...
  }
```

それから、`a` を続行 `k` に渡して新しいアクション `rb` を得ることができます:

```
ra >>= k = Reader (\e -> let a = runReader ra e
                     rb = k a
                     in ...)

def flatMap[A, B](ra: Reader[E, A])(k: A => Reader[E, B]): Reader[E, B] =
  Reader { e =>
    val a = runReader(ra)(e)
    val rb = k(a)
    ...
  }
```

最後に、環境 `e` と共にアクション `rb` を実行することができます:

```
ra >>= k = Reader (\e -> let a = runReader ra e
                     rb = k a
                     in runReader rb e)

def flatMap[A, B](ra: Reader[E, A])(k: A => Reader[E, B]): Reader[E, B] =
  Reader { e =>
    val a = runReader(ra)(e)
    val rb = k(a)
    runReader(rb)(e)
  }
```

`return` を実装するために、私たちは環境を無視して変更されていない値を返すアクションを作成します。

それをすべてまとめると、いくつかの単純化の後に、次の定義が得られます:

```
instance Monad (Reader e) where
    ra >>= k = Reader (\e -> runReader (k (runReader ra e)) e)
    return x = Reader (\e -> x)

implicit def readerMonad[E] = new Monad[Reader[E, ?]] {
    def pure[A](x: A): Reader[E, A] =
        Reader(e => x)

    def flatMap[A, B](ra: Reader[E, A])(k: A => Reader[E, B]): Reader[E, B] =
        Reader(e =>
            runReader(k(runReader(ra)(e)))(e))
}
```

21.2.4 書き込み専用状態

これは私たちの初期のロギングの例に過ぎません。飾り付けは `Writer` 関手によって与えられます:

```
newtype Writer w a = Writer (a, w)
```

```
case class Writer[W, A](run: (A, W))
```

完全性のために、データ構造をアンパックするための些細なヘルパー `runWriter` もあります：

```
runWriter :: Writer w a -> (a, w)
runWriter (Writer (a, w)) = (a, w)
```

```
def runWriter[W, A]: Writer[W, A] => (A, W) = {
    case Writer((a, w)) => (a, w)
}
```

私たちが以前に見たように、`Writer` を合成可能にするためには、`w` がモノイドでなければなりません。ここに `bind` 演算子に関して `Writer` のモナドインスタンスが書かれています：

```
instance (Monoid w) => Monad (Writer w) where
    (Writer (a, w)) >>= k = let (a', w') = runWriter (k a)
                                in Writer (a', w `mappend` w')
    return a = Writer (a, mempty)

implicit def writerMonad[W: Monoid] = new Monad[Writer[W, ?]] {
    def flatMap[A, B](wa: Writer[W, A])(k: A => Writer[W, B]): Writer[W, B] =
        wa match {
            case Writer((a, w)) =>
                val ((a1, w1)) = runWriter(k(a))
```

```
    Writer((a1, Monoid[W].combine(w, w1)))  
  }  
  
def pure[A](a: A) = Writer(a, Monoid[W].empty)  
}
```

21.2.5 状態

読み書きアクセスを持つ状態関数は、`Reader` と `Writer` の飾り付けを結合します。それらは純粋関数と考えることができます。それは追加の引数として状態を取り、結果として値／状態のペアを生成します: `(a, s) -> (b, s)`。Curry 化後、私たちはそれらを Kleisli 射の形式にしています `a -> (s -> (b, s))`、飾り付けは `State` 関手で抽象化されています:

```
newtype State s a = State (s -> (a, s))  
  
case class State[S, A](run: S => (A, S))
```

再び、私たちは Kleisli 射がアクションを返すと見なすことができます。それはヘルパー関数を使用して実行することができます:

```
runState :: State s a -> s -> (a, s)  
runState (State f) s = f s
```

```
def runState[S, A]: State[S, A] => S => (A, S) = {
    case State(f) => s => f(s)
}
```

異なる初期状態は、異なる結果だけでなく、異なる最終状態も生成するかもしれません。

`State` モナドのための bind の実装は、`Reader` モナドのそれと非常に似ていますが、各ステップで正しい状態を渡すための注意が必要です:

```
sa >>= k = State (\s -> let (a, s') = runState sa s
                      sb = k a
                      in runState sb s')
```

```
def flatMap[A, B](sa: State[S, A])(k: A => State[S, B]): State[S, B] =
  State { s =>
    val (a, s1) = runState(sa)(s)
    val sb = k(a)
    runState(sb)(s1)
}
```

ここに完全なインスタンスがあります:

```
instance Monad (State s) where
  sa >>= k = State (\s -> let (a, s') = runState sa s
                        in runState (k a) s')
```

```
    return a = State (\s -> (a, s))

implicit def stateMonad[S] = new Monad[State[S, ?]] {
  def flatMap[A, B](sa: State[S, A])(k: A => State[S, B]): State[S, B] =
    State { s =>
      val (a, s1) = runState(sa)(s)
      runState(k(a))(s1)
    }

  def pure[A](a: A): State[S, A] = State(s => (a, s))
}
```

状態を操作するために使用することができる 2 つのヘルパー Kleisli 射もあります。そのうちの一つは検査のために状態を取得します:

```
get :: State S s
get = State (\s -> (s, s))

def get: State[S, S] = State(s => (s, s))
```

そして、もう一つはそれを完全に新しい状態で置き換えます:

```
put :: s -> State s ()
put s' = State (\s -> ((), s'))

def put(s1: S): State[S, Unit] = State(s => (((), s1)))
```

21.2.6 例外

例外をスローする命令型の関数は、本当は部分関数です – それはその引数のいくつかの値に対して定義されていない関数です。純粋な全関数に関して例外を実装する最も単純な方法は **Maybe** 関手を使用することです。部分関数は、それが意味をなすときはいつでも **Just a** を返し、そうでないときは **Nothing** を返すように拡張された全関数になります。私たちが失敗の原因に関する情報も返したい場合、代わりに **Either** 関手を使用することができます (最初の型は例えば **String** に固定されます)。

ここに **Maybe** のための **Monad** インスタンスがあります:

```
instance Monad Maybe where
    Nothing >>= k = Nothing
    Just a >>= k = k a
    return a = Just a

implicit val optionMonad = new Monad[Option] {
    def flatMap[A, B](ma: Option[A])(k: A => Option[B]): Option[B] =
        ma match {
            case None => None
            case Some(a) => k(a)
        }

    def pure[A](a: A): Option[A] = Some(a)
}
```

`Maybe` のためのモナド的な合成が正しく計算をショートサーキットすることに注意してください(続行 `k` は決して呼ばれません)。それは私たちが例外から期待する振る舞いです。

21.2.7 繙続

これは「私たちがあなたを呼ぶので、あなたは私たちを呼ばないでください！」という状況です。あなたが仕事の面接の後に経験するかもしれません。直接の答えを得る代わりに、結果で呼ばれるべきハンドラ、つまり関数を提供することになります。このスタイルのプログラミングは、特に結果が呼び出しの時点で知られていない場合に有用です。例えば、それが別のスレッドで評価されたり、リモートのウェブサイトから配信される場合です。この場合の Kleisli 射は、ハンドラを受け入れる関数を返します。これは「計算の残りの部分」を表します:

```
data Cont r a = Cont ((a -> r) -> r)

case class Cont[R, A](run: (A => R) => R)
```

最終的に呼ばれるハンドラ `a -> r` は、最終結果型 `r` の結果を生成し、この結果は最後に返されます。継続は結果型によってパラメータ化されます。(実際には、これはしばしば何らかの種類の状態指標です。)

Kleisli 射が返すアクションを実行するためのヘルパー関数もあります。それはハンドラを取り、継続に渡します:

```
runCont :: Cont r a -> (a -> r) -> r
runCont (Cont k) h = k h

def runCont[R, A]: Cont[R, A] => (A => R) => R = {
  case Cont(k) => h => k(h)
}
```

継続の合成は名高く難しいので、モナドを通じた、そして特に `do` 表記を通じた処理は極めて有利です。

`bind` の実装を理解しましょう。まずはシンプルなシグネチャを見てみましょう:

```
(>>=) :: ((a -> r) -> r) ->
  (a -> (b -> r) -> r) ->
    ((b -> r) -> r)

def flatMap[A, B]: ((A => R) => R) =>
  (A => (B => R) => R) =>
    ((B => R) => R)
```

私たちの目標は、ハンドラ (`b -> r`) を取り、結果 `r` を生成する関数を作成することです。だからそれが私たちの出発点です:

```
ka >>= kab = Cont (\hb -> ...)
```

```
def flatMap[A, B](ka: Cont[R, A])(kab: A => Cont[R, B]): Cont[R, B] =  
  Cont(hb => ...)
```

ラムダの中で、私たちは適切なハンドラを持つ関数 `ka` を呼び出すことを望んでいます。このハンドラはラムダとして実装されます:

```
runCont ka (\a -> ...)
```

```
runCont(ka)(a => ...)
```

この場合、残りの計算にはまず `kab` で `a` を呼び出し、次に得られたアクション `kb` に `hb` を渡すことが含まれます:

```
runCont ka (\a -> let kb = kab a  
           in runCont kb hb)
```

```
runCont(ka) { a =>  
  val kb = kab(a)  
  runCont(kb)(hb)  
}
```

ご覧のとおり、継続は内側から外側に合成されます。最終的なハンドラ `hb` は計算の最も内側の層から呼び出されます。こちらが完全なインスタンスです:

```
instance Monad (Cont r) where
    ka >>= kab = Cont (\hb -> runCont ka (\a -> runCont (kab a) hb))
    return a = Cont (\ha -> ha a)

implicit def contMonad[R] = new Monad[Cont[R, ?]] {
    def flatMap[A, B](ka: Cont[R, A])(kab: A => Cont[R, B]): Cont[R, B] =
        Cont { hb =>
            runCont(ka)(a => runCont(kab(a))(hb))
        }

    def pure[A](a: A): Cont[R, A] =
        Cont(ha => ha(a))
}
```

21.2.8 対話型入力

これは最も厄介な問題で、多くの混乱の原因となっています。明らかに、`getChar` のような関数がキーボードで入力された文字を返す場合、それは純粋ではありません。しかし、それが文字をコンテナの中で返す場合はどうでしょうか？このコンテナから文字を抽出する方法がない限り、私たちはその関数が純粋だと主張することができます。あなたが `getChar` を呼び出すたびに、それは正確に同じコンテナを返します。概念的には、このコンテナはすべての可能な文字の重ね合わせを含んでいます。

もし量子力学に精通していれば、この類推を理解するのは問題ありません。それは、中を覗くことができないシュレーディンガーの猫が入っている箱のようなものです。箱は特別な組み込みの `IO` 関手を使用して定義されます。私たちの例では、`getChar` は Kleisli 射として宣言することができます:

```
getChar :: () -> IO Char
```

```
def getChar: Unit => IO[Char]
```

(実際には、`Unit` 型からの関数は、戻り型の値を選ぶことと等価であるため、`getChar` の宣言は `getChar :: IO Char` に簡略化されます。)

関手である `IO` は、`fmap` を使用してその内容を操作することを可能にします。そして、関手として、それは文字だけでなく、任意の型の内容を格納することができます。このアプローチの真の有用性は、Haskell で `IO` がモナドであると考えたときに明らかになります。それは、`IO` オブジェクトを生成する Kleisli 射を合成することができることを意味します。

Kleisli 合成が `IO` オブジェクトの内容をのぞき見することを可能にするとと思うかもしれません (量子論のアナロジーを続けるならば、「波動関数の崩壊」)。確かに、`getChar` を別の Kleisli 射、例えば文字を整数に変換するものと合成することができます。しかし、この第二の Kleisli 射は、この整数を (`IO Int`) として返すことしかできません。再び、あなたはすべての可能な整数の重ね合わせで終わるでしょう。そ

して、以下に続きます。シュレーディンガーの猫は決して袋から出ません。一度 `IO` モナドの中にいると、それから抜け出す方法はありません。`runState` や `runReader` に相当するものは、`IO` モナドにはありません。`runIO` はありません！

では、もう一つの Kleisli 射でそれを合成する以外に、Kleisli 射の結果、`IO` オブジェクトで何ができるでしょうか？ まあ、それを `main` から返すことができます。Haskell では、`main` は次のシグネチャを持っています：

```
main :: IO ()  
  
def main: IO[Unit]
```

そして、それを Kleisli 射として考える自由があります：

```
main :: () -> IO ()  
  
def main: Unit => IO[Unit]
```

その観点から、Haskell プログラムは単に `IO` モナドでの 1 つの大きな Kleisli 射です。あなたはそれを小さな Kleisli 射からモナド的な合成を使用して構成することができます。それを何かするのはランタイムシステムに任されています（また、`IO` アクションと呼ばれます）。

射自体が純粋関数であることに注意してください — それは純粋関数が一番下まで続いている。汚れた仕事はシステムに委ねられます。

それが `main` から返された `IO` アクションを最終的に実行するとき、それはあらゆる種類のいやらしいことをします。例えば、ユーザー入力の読み取り、ファイルの変更、不快なメッセージの印刷、ディスクのフォーマットなどです。Haskell プログラムは決して自分の手を汚しません（まあ、それが `unsafePerformIO` を呼び出さない限りですが、それは別の話です）。

もちろん、Haskell が怠惰であるため、`main` はほぼ即座に戻り、汚れた仕事がすぐに始まります。純粋な計算の結果が要求され、必要に応じて評価されるのは、`IO` アクションの実行中です。したがって、実際にには、プログラムの実行は純粋な (Haskell) コードと汚れた (システム) コードの相互作用です。

`IO` モナドのもう一つの解釈は、さらに奇妙ですが、数学的モデルとして完全に理にかなっています。それはプログラム内のオブジェクトとして宇宙全体を扱うことです。概念的には、命令型モデルは宇宙を外部のグローバルオブジェクトとして扱うので、I/O を実行する手続きはそのオブジェクトとの相互作用によって副作用を持っています。彼らは宇宙の状態を読み取り、変更することができます。

私たちは既に、関数型プログラミングで状態をどのように扱うかを知っています — 私たちは状態モナドを使用します。しかし、宇宙の状態は単純な状態とは異なり、標準的なデータ構造を使用して簡単に記述することはできません。しかし、私たちがそれと直接的に相互作用しない限り、私たちはそれをしなくとも済みます。存在すると仮定するだけで十分です、型 `RealWorld` があり、宇宙工学の何らかの奇跡に

よって、ランタイムがこの型のオブジェクトを提供することができる
と。IO アクションは単なる関数です:

```
type IO a = RealWorld -> (a, RealWorld)
```

```
type IO[A] = RealWorld => (A, RealWorld)
```

または State モナドの用語で:

```
type IO = State RealWorld
```

```
type IO[A] = State[RealWorld, A]
```

ただし、IO モナドの `>=>` と `return` は言語に組み込まれていなければ
なりません。

21.2.9 対話型出力

同じ IO モナドは、対話型出力をカプセル化するために使用されます。`RealWorld` はすべての出力デバイスを含むとされています。なぜ
私たちは単に Haskell から出力関数を呼び出し、それらが何もしない
ふりをすることができないのか疑問に思うかもしれません。例えば、
なぜ私たちは:

```
putStr :: String -> IO ()  
  
def putStr: String => IO[Unit]
```

を持つのではなく、より単純な:

```
putStr :: String -> ()  
  
def putStr: String => Unit
```

を持たないのでしょうか。二つの理由があります: Haskell は怠惰なので、使用されない出力 – ここでは単位オブジェクト – を持つ関数を決して呼び出しません。そして、怠惰でなかったとしても、それはまだそのような呼び出しの順序を自由に変更し、出力をめちゃくちゃにすることができます。Haskell で二つの関数の順次実行を強制する唯一の方法は、データ依存関係を通じてです。一方の関数の入力は、他方の関数の出力に依存する必要があります。RealWorld を IO アクション間で渡すことによって、シーケンシングが強制されます。

概念的には、このプログラムでは:

```
main :: IO ()  
main = do  
    putStrLn "Hello "  
    putStrLn "World!"
```

```
def main: IO[Unit] = for {
    _ <- putStrLn("Hello ")
    _ <- putStrLn("World!")
} yield ()
```

「World!」を印刷するアクションは、「Hello」がすでに画面にある宇宙を入力として受け取ります。それは画面に「Hello World!」がある新しい宇宙を出力として出力します。

21.3 結論

もちろん、私はモナド的プログラミングの表面をかすめただけです。モナドは、命令型プログラミングで通常副作用を使用して行われることを純粋関数で達成するだけでなく、高度な制御と型安全性を提供します。しかし、それらには欠点もあります。モナドに関する主な不満の一つは、それらが互いに容易に合成されないことです。確かに、モナド変換ライブラリを使用して、基本的なモナドのほとんどを組み合わせることができます。たとえば、状態と例外を組み合わせたモナドスタックを作成するのは比較的簡単ですが、任意のモナドと一緒に積み重ねるための公式はありません。

22

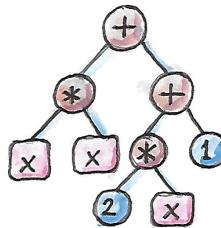
モナドを圏論的に

プログラマにモナドを話すと、作用について話すことになるかも
されませんが、数学者にとってモナドは代数についてです。後ほど代数について話します — プログラミングにおいて重要な役割を果たします — しかしそれ、モナドとの関係について少し直感をお伝えしたいです。今は少し手を振るような議論ですが、お付き合いください。

代数は、表現を作成し、操作し、評価することについてです。表現は演算子を使って構成されます。このシンプルな表現を考えてみてください:

$$x^2 + 2x + 1$$

この表現は、変数 x や定数 1 や 2 を、足し算や掛け算のような演算子で結びつけて構成されています。プログラマとしては、表現を木構造として考えることがよくあります。



木はコンテナなので、一般的には、表現は変数を格納するコンテナです。圏論では、コンテナを自己関手として表します。変数 x に型 a を割り当てると、私たちの表現は ma という型を持ちます。ここで m は、表現木を構成する自己関手です。(非自明な分岐表現は通常、再帰的に定義された自己関手を使用して作成されます。)

表現に対して実行できる最も一般的な操作は何でしょうか？ それは代入です：変数を表現で置き換えることです。例えば、先ほどの例で、 X を $y - 1$ で置き換えると、以下のようになります：

$$(y - 1)^2 + 2(y - 1) + 1$$

ここで何が起こったかというと、型 ma の表現を取り、型 $a \rightarrow mb$ の変換 (b は y の型) を適用しました。結果は型 mb の表現です。はっきりさせましょう：

$$ma \rightarrow (a \rightarrow mb) \rightarrow mb$$

はい、これがモナド的な束縛のシグネチャです。

これは少し動機付けです。さあ、モナドの数学に取り組みましょう。数学者はプログラマよりも異なる記法を好むことがあります。彼らは自己関手には文字 T を使用し、ギリシャ文字: μ は **join** に、 η は **return** に対応します。両方の **join** と **return** は多相的な関数なので、それらが自然変換に対応すると推測できます。

したがって、圏論では、モナドはペアの自然変換 μ と η を備えた自己関手 T として定義されます。

μ は、関手 T^2 から T に戻る自然変換です。 T^2 は単に関手自体との自体の組み合わせ、 $T \circ T$ です (この種の二乗は自己関手に対してのみ実行できます)。

$$\mu :: T^2 \rightarrow T$$

この自然変換の対象 a におけるコンポーネントは射です:

$$\mu_a :: T(Ta) \rightarrow Ta$$

これは **Hask** では **join** の定義に直接翻訳されます。

η は、恒等関手 I と T の間の自然変換です:

$$\eta :: I \rightarrow T$$

I の対象 a に対する作用が単に a であることを考慮すると、 η のコンポーネントは射によって与えられます:

$$\eta_a :: a \rightarrow Ta$$

これは `return` の定義に直接翻訳されます。

これらの自然変換はいくつかの追加の規則を満たさなければなりません。一つの見方は、これらの規則が私たちに自己関手 T のための Kleisli 圏を定義させることです。 a と b の間の Kleisli 射は、射 $a \rightarrow Tb$ として定義されます。2つのそのような射の合成 (T の下付き文字を持つ円として書きます) は μ を使用して実装できます:

$$g \circ_T f = \mu_c \circ (Tg) \circ f$$

ここで

$$\begin{aligned}f &:: a \rightarrow Tb \\g &:: b \rightarrow Tc\end{aligned}$$

ここで T は関手であり、射 g に適用できます。この式を Haskell 記法で認識するのが容易かもしれません:

```
f >=> g = join . fmap g . f

(f >=> g) ==
  (flatten compose fmap[B, T[C]](g) compose f)
```

または、コンポーネントで:

```
(f >=> g) a = join (fmap g (f a))
```

```
(f >=> g)(a) == flatten(fmap(g)(f(a)))
```

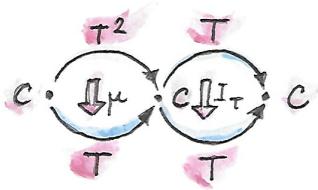
代数的な解釈では、私たちは単に 2 つの連続する代入を合成しているだけです。

Kleisli 射が圏を形成するためには、その合成が結合的であること、そして η_a が a での恒等 Kleisli 射であることが望まれます。この要件は、 μ と η のためのモナドの規則に翻訳することができます。しかし、これらの規則をよりモノイドの規則のように見せる別の方法で導出することもできます。実際 μ はしばしば**乗算**と呼ばれ、 η は**単位**と呼ばれます。

大まかに言えば、結合則は T の立方体、 T^3 、を T に減らす 2 つの方法が同じ結果を与えることになります。2 つの単位則（左と右）は、 η が T に適用されてから μ によって減少されるとき、私たちは T に戻ると述べています。

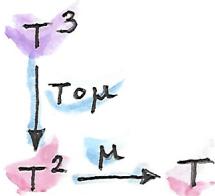
これらの規則は少しトリッキーです、なぜなら自然変換と関手を合成しているからです。そのため、水平合成について少し復習が必要です。例えば、 T^3 は T の後に T^2 の合成と見なすことができます。私たちはそれに 2 つの自然変換の水平合成を適用することができます:

$$I_T \circ \mu$$

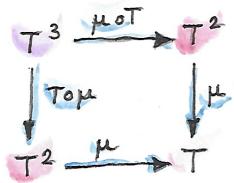


そして $T \circ T$ を得ることができます；これは μ を適用することさらに T に減少できます。 I_T は T から T への恒等自然変換です。この種類の水平合成の表記 $I_T \circ \mu$ を $T \circ \mu$ に短縮するのをよく見ます。この表記は曖昧ではありません、なぜなら関手と自然変換を合成することは意味がないので、 T はこの文脈で I_T を意味しなければなりません。

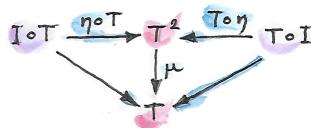
また、(自己) 関手圏 $[C, C]$ における図を描くことができます：



代わりに、 T^3 を $T^2 \circ T$ の合成として扱い、それに $\mu \circ T$ を適用することもできます。結果はまた $T \circ T$ で、再び μ を使って T に減少することができます。私たちは 2 つのパスが同じ結果を生むことを要求します。



同様に、私たちは恒等関手 I の後に T の合成に水平合成 $\eta \circ T$ を適用して T^2 を得ることができ、それから μ を使用して減少することができます。結果は、恒等自然変換を直接 T に適用した場合と同じでなければなりません。同様に、 $T \circ \eta$ にも真実でなければなりません。



これらの規則が Kleisli 射の合成が確かに圏の規則を満たすことを保証することを自分自身に納得させることができます。

モナドとモノイドの類似性は著しいです。私たちは乗算 μ 、単位 η 、結合則、および単位則を持っています。しかし、モナドをモノイドとして記述するためには、私たちのモノイドの定義が狭すぎます。では、モノイドの概念を一般化しましょう。

22.1 モノイダル圏

モノイドの従来の定義に立ち戻りましょう。それは二項演算と単位と呼ばれる特別な要素を持つ集合です。Haskell では、これは型クラスとして表現されます:

```
class Monoid m where
    mappend :: m -> m -> m
    mempty  :: m

trait Monoid[M] {
    def combine(x: M, y: M): M
    def empty: M
}
```

二項演算 `mappend` は結合的であり、単位的でなければなりません(つまり、単位 `mempty` による乗算は無操作です)。

Haskell での `mappend` の定義は Curry 化されています。それはすべての `m` の要素を関数にマッピングすると解釈することができます:

```
mappend :: m -> (m -> m)

def combine(x: M): (M => M)
```

この解釈は、モノイドをモノイドの要素として自己準同型 ($m \rightarrow m$) を表す单一対象圏として定義する原因となります。しかし、Curry 化

が Haskell に組み込まれているため、私たちは乗算の異なる定義から始めることもできました:

```
mul :: (m, m) -> m
```

```
def mul: ((M, M)) => M
```

ここで、デカルト積 (m, m) は乗算されるペアのソースになります。

この定義は、圏論的な積でデカルト積を置き換えることによって一般化への別の道を示唆しています。積が大局的に定義された圏を始めることができ、対象 m を選んで、乗算を射として定義することができます:

$$\mu : m \times m \rightarrow m$$

しかし、任意の圏で対象の内部を見ることはできないため、単位要素をどのように選ぶかという問題があります。そのためにはトリックがあります。要素選択が单集合からの関数と同等であることを覚えていましたか？ Haskell では、`mempty` の定義を関数で置き換えることができます:

```
eta :: () -> m
```

```
def eta: Unit => M
```

単集合は **Set** における終対象なので、終対象 t を持つ任意の圏にこの定義を一般化することは自然です:

$$\eta :: t \rightarrow m$$

これにより、要素について話すことなく、単位「要素」を選ぶことができます。

以前のモノイドとしての单一対象圏の定義とは異なり、ここではモノイダル則は自動的に満たされません—私たちはそれらを課さなければなりません。しかし、それらを定式化するためには、基礎となる圏論的な積自体のモノイダル構造を確立する必要があります。まず Haskell でのモノイダル構造がどのように機能するかを思い出しましょう。

結合性から始めます。Haskell で、対応する等式は次のようにになります:

$$\text{mu } (x, \text{mu } (y, z)) = \text{mu } (\text{mu } (x, y), z)$$

$$\text{mu } (x, \text{mu } (y, z)) == \text{mu } (\text{mu } (x, y), z)$$

他の圏に一般化する前に、個々の変数に対するその作用からそれを書き換えて、関数(射)の等式として書く必要があります。言い換えると、それをポイントフリー表記で使用する必要があります。デカルト積が双関手であることを知っていれば、左辺を次のように書くことができます:

```
(mu . bimap id mu)(x, (y, z))
```

```
mu compose bimap(identity)(mu)
```

右辺は次のようにになります:

```
(mu . bimap mu id)((x, y), z)
```

```
mu compose bimap(mu)(identity)
```

これはほぼ私たちが望むものです。残念ながら、デカルト積は厳密に結合的ではない— $(x, (y, z))$ は $((x, y), z)$ と同じではありません—したがって、ポイントフリーで書くことはできません:

```
mu . bimap id mu = mu . bimap mu id
```

```
mu.compose(bimap(identity)(mu)) ==
mu.compose(bimap(mu)(identity))
```

他方で、ペアの 2 つの入れ子は同型です。それらの間を変換する可逆関数、結合子 (associator) があります:

```
alpha :: ((a, b), c) -> (a, (b, c))
alpha ((x, y), z) = (x, (y, z))
```

```
def alpha[A, B, C]: (((A, B), C)) => ((A, (B, C))) = {
    case ((x, y), z) => (x, (y, z))
}
```

結合子の助けを借りて、`mu` のポイントフリーな結合則を書くことができます:

```
mu . bimap id mu . alpha = mu . bimap mu id

mu.compose(
    bimap(identity)(mu) compose alpha) ==
mu.compose(bimap(mu)(identity))
```

同様のトリックを単位則に適用することができます。新しい表記では、次の形を取ります:

```
mu (eta (), x) = x
mu (x, eta ()) = x

mu(eta(), x) == x
mu(x, eta()) == x
```

それらは次のように書き換えることができます:

```
(mu . bimap eta id) ((), x) = lambda((), x)
(mu . bimap id eta) (x, ()) = rho (x, ())
```

```
mu.compose(bimap(eta)(identity[M]))((((), x)) == lambda((((), x))
```

```
mu.compose(bimap(identity[M])(eta))((x, ()) == rho((x, ()))
```

同型 `lambda` と `rho` はそれぞれ左単位子と右単位子と呼ばれます。これらは、単位 `()` が同型の違いを除いてデカルト積の恒等写像であるという事実を示しています:

```
lambda :: (((), a) -> a  
lambda (((), x)) = x
```

```
def lambda[A]: ((Unit, A)) => A = {  
  case (((), x)) => x  
}
```

```
rho :: (a, ()) -> a  
rho (x, ()) = x
```

```
def rho[A]: ((A, Unit)) => A = {  
  case (x, ()) => x  
}
```

したがって、単位則のポイントフリー版は次のようにになります:

```
mu . bimap id eta = rho  
mu . bimap eta id = lambda
```

```

mu.compose(bimap(eta)(identity[M])) == lambda
mu.compose(bimap(identity[M])(eta)) == rho

```

私たちは、基礎となるデカルト積自体が型の圏でモノイダル乗算のように機能するという事実を使用して、`mu` と `eta` のポイントフリーなモノイダル則を定式化しました。ただし、デカルト積の結合性と単位則は、同型によってのみ有効です。

これらの規則は、積と終対象を持つ任意の圏に一般化することができます。圏論的な積は、実際には同型によって結合的であり、終対象は同型によって単位です。結合子と 2 つの単位子は自然同型です。規則は可換図式によって表されます。

$$\begin{array}{ccc}
 ((\alpha \times \alpha) \times \alpha & \xrightarrow{\alpha} & \alpha \times (\alpha \times \alpha) \\
 \downarrow \mu \times id & & \downarrow id \times \mu \\
 \alpha \times \alpha & \xrightarrow{\mu} & \alpha \times \alpha \\
 & \searrow \mu & \swarrow \mu \\
 & a &
 \end{array}$$

積が双関手であるため、対の射を持ち上げることができます – Haskell ではこれは `bimap` を使用して行われました。

ここで止めて、圏論的な積と終対象を持つ任意の圏の上にモノイドを定義することができると言うことができます。対象 m と 2 つの射 μ と η を選び、モノイダル則を満たすことができれば、モノイドを持っています。しかし、私たちはそれよりも良くすることができます。 μ

と η の規則を定式化するために完全な圏論的な積を必要とするわけではありません。積が射影を使用する普遍構成を通じて定義されることを思い出してください。私たちのモノイダル則の定式化では、射影を使用していません。

積のように振る舞うが積ではない双関手は、しばしば中置演算子 \otimes で表される**テンソル積**と呼ばれます。一般にテンソル積の定義は少し難しいですが、私たちにはそれについて心配しません。私たちにはその特性をリストするだけです – 最も重要なのは同型による結合性です。

同様に、対象 t が終対象である必要はありません。その終対象の性質、つまりそれへの任意の対象からの一意な射の存在を使用したことではありません。私たちが必要とするのは、それがテンソル積と連携して機能することです。つまり、同型によってテンソル積の単位であることが望されます。全体をまとめましょう:

モノイダル圏は、テンソル積と呼ばれる双関手を備えた圏 \mathbf{C} です:

$$\otimes : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}$$

そして、単位対象 i と呼ばれる特異な対象、およびそれぞれ結合子と左右の単位子と呼ばれる 3 つの自然同型があります:

$$\alpha_{abc} :: (a \otimes b) \otimes c \rightarrow a \otimes (b \otimes c)$$

$$\lambda_a :: i \otimes a \rightarrow a$$

$$\rho_a :: a \otimes i \rightarrow a$$

(四重テンソル積を単純化するための整合性条件もあります。)

重要なのは、テンソル積が多くのおなじみの双関手を記述することです。特に、積、余積、およびまもなく見るように、自己関手の合成(およびDay畠み込みのようないくつかのより奇妙な積)に対して機能します。モノイダル圏は、豊穣圏の定式化において重要な役割を果たします。

22.2 モノイダル圏におけるモノイド

私たちは今、モノイダル圏のより一般的な設定でモノイドを定義する準備ができました。対象 m を選びます。テンソル積を使用して m の累乗を形成できます。 m の二乗は $m \otimes m$ です。 m の立方体を形成する2つの方法がありますが、結合子を介して同型です。同様に、 m のより高い累乗のために(整合性条件が必要です)。モノイドを形成するために、2つの射を選びます:

$$\mu :: m \otimes m \rightarrow m$$

$$\eta :: i \rightarrow m$$

ここで i は私たちのテンソル積のための単位対象です。



これらの射は結合則と単位則を満たさなければなりませんが、それらは次の可換図式の形で表現できます:

$$\begin{array}{ccc}
 (m \otimes m) \otimes m & \xrightarrow{\alpha} & m \otimes (m \otimes m) \\
 \downarrow \mu \otimes id & & \downarrow id \otimes \mu \\
 m \otimes m & \xrightarrow{\mu} & m \otimes m \\
 & \downarrow \mu & \downarrow \mu \\
 & m &
 \end{array}$$

$$\begin{array}{ccccc}
 i \otimes m & \xrightarrow{\eta \otimes id} & m \otimes m & \xleftarrow{id \otimes \eta} & m \otimes i \\
 & \searrow \lambda & \downarrow \mu & \swarrow \beta & \\
 & m & & m &
 \end{array}$$

テンソル積が双関手であることが重要です。なぜなら、私たちは射のペアを持ち上げて $\mu \otimes id$ や $\eta \otimes id$ のような積を形成する必要がある

からです。これらの図式は、圏論的な積に対する私たちの以前の結果の直接的な一般化です。

22.3 モナドとしてのモノイド

モノイダル構造は予期せぬ場所で現れます。その一つが関手圏です。少し目を細めれば、関手の合成が乗法の一種として見えるかもしれません。問題は、任意の 2 つの関手が合成可能であるわけではないことです — 一つの関手の目的圏がもう一つの関手の出発圏でなければなりません。これは射の合成に関する通常の規則です — そして、私たちは関手が実際には圏 Cat の射であることを知っています。しかし、自己準同型 (同じ対象に戻る射) は常に合成可能であるように、自己関手もそうです。任意の与えられた圏 C の自己関手は、関手圏 $[C, C]$ を形成します。その対象は自己関手であり、射はそれらの間の自然変換です。この圏から任意の 2 つの対象、例えば自己関手 F と G を取り、それらの合成である第三の対象 $F \circ G$ — それらの合成である自己関手を生成できます。

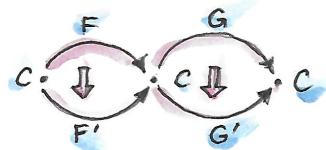
自己関手の合成はテンソル積の良い候補ですか？ まず、それが双関手であることを確立する必要があります。ここでの射のペア — つまり、自然変換 — を持ち上げることができますか？ テンソル積の **bimap** の類似物の署名は次のようになります：

$$\text{bimap} :: (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow (a \otimes c \rightarrow b \otimes d)$$

対象を自己関手に、矢を自然変換に、そしてテンソル積を合成に置き換えると、次のようになります:

$$(F \rightarrow F') \rightarrow (G \rightarrow G') \rightarrow (F \circ G \rightarrow F' \circ G')$$

これは水平合成の特別なケースとして認識できます。



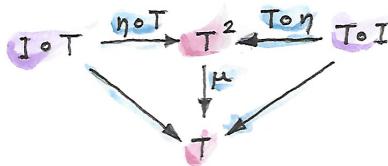
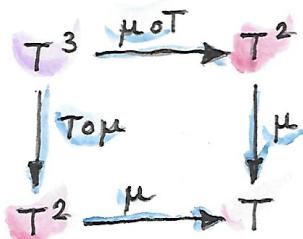
また、私たちの新しいテンソル積のための恒等写像として機能できる、恒等自己関手 I も利用可能です。さらに、関手の合成は結合的です。実際、結合則と単位則は厳格です – 結合子や 2 つの単位子はありません。したがって、自己関手は、関手の合成をテンソル積として、厳密なモノイダル圏を形成します。

この圏でのモノイドとは何ですか？ それは対象 – つまり自己関手 T ；そして 2 つの射 – つまり自然変換です：

$$\mu :: T \circ T \rightarrow T$$

$$\eta :: I \rightarrow T$$

それだけでなく、ここにモノイド則があります:



それらは、以前見たモナド則とまさに同じです。これで、Saunders Mac Lane の有名な引用を理解できます：

総じて、モナドは単に自己関手の圏のモノイドです。

あなたはそれを機能的プログラミング会議でいくつかの T シャツに掲げられているかもしれません。

22.4 隨伴からのモナド

随伴 $L \dashv R$ は、2 つの圏 C と D の間を行き来する一対の関手です。それらを合成する 2 つの方法があり、2 つの自己関手 $R \circ L$ と $L \circ R$ が生まれます。随伴によれば、これらの自己関手は単位と余単位と呼ば

れる 2 つの自然変換を介して恒等関手に関連しています:

$$\begin{aligned}\eta &:: I_D \rightarrow R \circ L \\ \varepsilon &:: L \circ R \rightarrow I_C\end{aligned}$$

直ちに、随伴の単位はモナドの単位とまったく同じように見えることがわかります。実際、自己関手 $R \circ L$ はモナドです。適切な μ を η とともに定義するだけでよいのです。それは、私たちの自己関手の二乗と自己関手自体の間の自然変換です、または随伴関手の観点からは:

$$R \circ L \circ R \circ L \rightarrow R \circ L$$

実際、私たちは中間の $L \circ R$ を崩壊させるために余単位を使用することができます。 μ の正確な式は、水平合成によって与えられます:

$$\mu = R \circ \varepsilon \circ L$$

モナド則は、随伴の単位と余単位によって満たされる恒等式と交換則から導かれます。

Haskell では、随伴が 2 つの圏に関与するため、随伴から派生したモナドをあまり見ません。ただし、指數関数、または関数対象の定義は例外です。この随伴を形成する 2 つの自己関手は次のようにになります:

$$Lz = z \times s$$

$$Rb = s \Rightarrow b$$

その合成を、おなじみの State モナドとして認識するかもしれません:

$$R(Lz) = s \Rightarrow (z \times s)$$

これは、以前に Haskell で見たモナドです:

```
newtype State s a = State (s -> (a, s))  
  
case class State[S, A](run: S => (A, S))
```

随伴を Haskell に翻訳しましょう。左関手は Product 関手です:

```
newtype Prod s a = Prod (a, s)  
  
case class Prod[S, A](run: (A, S))
```

そして、右関手は Reader 関手です:

```
newtype Reader s a = Reader (s -> a)  
  
case class Reader[S, A](run: S => A)
```

それらは随伴を形成します:

```
instance Adjunction (Prod s) (Reader s) where  
  counit (Prod (Reader f, s)) = f s  
  unit a = Reader (\s -> Prod (a, s))  
  
implicit def state[S] = new Adjunction[Prod[S, ?], Reader[S, ?]] {  
  def counit[A](a: Prod[S, Reader[S, A]]): A = a match {  
    case Prod((Reader(f), s)) => f(s)
```

```
}

def unit[A](a: A): Reader[S, Prod[S, A]] =
  Reader(s => Prod((a, s)))

}
```

Product 関手の後の Reader 関手の合成が確かに State 関手に等しいことを簡単に納得することができます:

```
newtype State s a = State (s -> (a, s))

case class State[S, A](run: S -> (A, S))
```

予想通り、随伴の `unit` は State モナドの `return` 関数に相当します。余単位は、その引数に作用する関数を評価することで作用します。これは `runState` の未 Curry 化バージョンとして認識できます:

```
runState :: State s a -> s -> (a, s)
runState (State f) s = f s

def runState[S, A]: State[S, A] => S -> (A, S) = {
  case State(f) => s => f(s)
}
```

(未 Curry 化、なぜなら `counit` はペアに作用するからです)。

私たちは、3つの自然変換の水平合成を必要とする自然変換 μ の一部として State モナドの `join` を定義することができます:

$$\mu = R \circ \varepsilon \circ L$$

つまり、余単位 ε を Reader 関手の 1 レベルを超えて潜り込ませる必要があります。コンパイラは `Reader` 関手ではなく `State` 関手のための `fmap` を選択するため、直接 `fmap` を呼び出すことはできません。しかし、Reader 関手の `fmap` は左関数合成に過ぎないことを思い出してください。だから、関数合成を直接使用します。

まず、`State` データ構造を剥がして、`State` 関手内の関数を露出させる必要があります。これは `runState` を使用して行われます:

```
ssa :: State s (State s a)
runState ssa :: s -> (State s a, s)
```

```
def ssa[S, A]: State[S, State[S, A]]
```

```
def rss[S, A]: S => (State[S, A], S) =
  runState[S, State[S, A]](ssa)
```

次に、それを余単位である `uncurry runState` で左合成します。最後に、それを `State` データ構造で再び覆います:

```
join :: State s (State s a) -> State s a
join ssa = State (uncurry runState . runState ssa)

def join[S, A]: State[S, State[S, A]] => State[S, A] =
  ssa => {
    State(
      Function.uncurried(runState[S, A])
      .tupled
      .compose(runState(ssa))
    )
  }
}
```

これは実際には、**State** モナドの **join** の実装です。

実際には、すべての随伴がモナドを生み出すだけでなく、逆もまた真であることがわかります: すべてのモナドは 2 つの随伴関手の合成に分解することができます。ただし、そのような分解は一意ではありません。

次のセクションで他の自己関手 $L \circ R$ について話します。

23

余モナド

モ ナドを理解した今、射を反転させ逆圏で作業することで、双対性の恩恵を受けて余モナドを自由に得ることができます。

基本的に、モナドは Kleisli 射の合成に関するものです:

a → m b

A => M[B]

ここで、m はモナドである関手です。余モナドについては、文字 w (m の逆さ) を使い、余 Kleisli 射を以下の型の射として定義できます:

```
w a -> b
```

```
W[A] => B
```

余 Kleisli 射のための魚オペレータの類似物は以下のように定義されます:

```
(=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
```

```
def =>=[A, B, C](w1: W[A] => B)(w2: W[B] => C): W[A] => C
```

余 Kleisli 射が圏を形成するためには、**extract** と呼ばれる恒等的な余 Kleisli 射も必要です:

```
extract :: w a -> a
```

```
def extract[A](wa: W[A]): A
```

これは **return** の双対です。また、結合則と左右の恒等則を課す必要があります。すべてをまとめると、Haskell で余モナドを定義することができます:

```
class Functor w => Comonad w where
  (=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
  extract :: w a -> a
```

```
trait Comonad[W[_]] extends Functor[W] {  
    def =>=[A, B, C](w1: W[A] => B)(w2: W[B] => C): W[A] => C  
    def extract[A](wa: W[A]): A  
}
```

実際には、後ほど見るように、わずかに異なる基本要素を使用します。
では、プログラミングにおける余モナドの使用法は何でしょうか？

23.1 余モナドでのプログラミング

モナドと余モナドを比較してみましょう。モナドは `return` を使って値をコンテナに入れる方法を提供します。それはコンテナ内に保存された値や値へのアクセスを提供しません。もちろん、モナドを実装するデータ構造はその内容にアクセスを提供するかもしれませんが、それはボーナスと考えられます。モナドから値を抽出する共通のインターフェースはありません。そして、その内容を決して公開しない `IO` モナドの例を見てきました。

一方、余モナドはそこから单一の値を抽出する手段を提供します。それは値を挿入する手段を提供しません。ですから、余モナドをコンテナと考えるなら、それは常に内容で満たされており、中身を覗くことができます。

Kleisli 射が値を取り、何らかの文脈を加えた結果を生成するように – それは文脈を加えることです – 余 Kleisli 射は文脈全体と一緒に値を取り、結果を生成します。それは文脈的計算の体現です。

23.2 積型余モナド

Reader モナドを覚えていますか？ 読み取り専用環境 e へのアクセスが必要な計算を実装するためにそれを導入しました。そのような計算は以下の形式の純粋関数として表されます：

$(a, e) \rightarrow b$

$((A, E)) \Rightarrow B$

私たちは Curry 化を使ってそれらを Kleisli 射に変換しました：

$a \rightarrow (e \rightarrow b)$

$A \Rightarrow (E \Rightarrow B)$

しかし、これらの関数はすでに余 Kleisli 射の形をしています。それらの引数をより便利な関手形式に整理しましょう：

```
data Product e a = Prod e a deriving Functor
```

```
case class Product[E, A](run: (E, A))
```

```
// implicit def productFunctor = ...
```

合成オペレータを簡単に定義できます。合成している射に同じ環境を利用可能にします:

```
(=>=) :: (Product e a -> b) -> (Product e b -> c) -> (Product e a -> c)
f =>= g = \(Prod e a) -> let b = f (Prod e a)
                           c = g (Prod e b)
                           in c

def =>=[E, A, B, C]: (Product[E, A] => B) => (Product[E, B] => C) =>
  (Product[E, A] => C) = f => g => {
  case Product((e, a)) =>
    val b = f(Product((e, a)))
    val c = g(Product((e, b)))
    c
}
```

`extract` の実装は単純に環境を無視します:

```
extract (Prod e a) = a

def extract[E, A]: Product[E, A] => A = {
  case Product((e, a)) => a
}
```

驚くべきことに、積型余モナドは Reader モナドとまったく同じ計算を実行するために使用することができます。ある意味で、余モナドの

環境実装はより自然です – それは「文脈における計算」の精神に従います。一方、モナドは `do` 表記の便利な構文を持っています。

Reader モナドと積型余モナドの間のつながりはさらに深く、Reader 関手が積関手の右随伴であるという事実に関係しています。一般的には、余モナドはモナドとは異なる計算の概念をカバーします。後でさらに多くの例を見るでしょう。

任意の積型、タプルやレコードを含む、**Product** 余モナドを一般化するのは簡単です。

23.3 合成の解剖

双対化のプロセスを続けると、モナド的な結合や `join` を双対化することができます。代わりに、魚オペレータの解剖に使用したプロセスを繰り返すことができます。このアプローチはより啓蒙的に思えます。

出発点は、合成オペレータが `w a` を取り、`c` を生成する余 Kleisli 射を生成する必要があるという実現です。唯一の方法で `c` を生成するには、`w b` 型の引数に第二の関数を適用することです:

```
(=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
f =>= g = g ...
```

```
def =>=[W[_], A, B, C]: (W[A] => B) => (W[B] => C) => (W[A] => C) =
  f => g => g ...
```

しかし、どのようにして $w\ b$ 型の値を生成し、それを g に供給することができるでしょうか？私たちが使用できるのは、型 $w\ a$ の引数と関数 $f :: w\ a \rightarrow b$ です。その解決策は、結合の双対である `extend` を定義することです：

```
extend :: (w a -> b) -> w a -> w b
```

```
def extend[W[_], A, B]: (W[A] => B) => W[A] => W[B]
```

`extend` を使って合成を実装できます：

```
f =>= g = g . extend f
```

```
def =>=[W[_], A, B, C]: (W[A] => B) => (W[B] => C) => (W[A] => C) =  
f => g => g compose extend(f)
```

次に、`extend` を解剖できるかどうかを考えるかもしれません。関数 $w\ a \rightarrow b$ を引数 $w\ a$ に適用するだけでいいのではないかと誘惑されるかもしれません、その結果の b を $w\ b$ に変換する方法がないことにすぐ気づきます。覚えておいてください、余モナドは値を持ち上げる手段を提供しません。この点で、モナドの類似構造では `fmap` を使いました。ここで `fmap` を使う唯一の方法は、 $w(w\ a)$ 型のものを手元に持っている場合です。もし私たちが $w\ a$ を $w(w\ a)$ に変換できればいいのです。そして、便利なことに、それはまさに `join` の双対です。それを `duplicate` と呼びます：

```
duplicate :: w a -> w (w a)
```

```
def duplicate[W[_], A]: W[A] => W[W[A]]
```

したがって、モナドの定義と同様に、余モナドには余 Kleisli 射、`extend`、または `duplicate` を使用する 3 つの同等の定義があります。以下は、`Control.Comonad` ライブラリから直接取られた Haskell の定義です：

```
class Functor w => Comonad w where
    extract :: w a -> a
    duplicate :: w a -> w (w a)
    duplicate = extend id
    extend :: (w a -> b) -> w a -> w b
    extend f = fmap f . duplicate

trait Comonad[W[_]] extends Functor[W] {
    def extract[A](wa: W[A]): A

    def duplicate[A](wa: W[A]): W[W[A]] =
        extend(identity[W[A]])(wa)

    def extend[A, B](f: W[A] => B)(wa: W[A]): W[B] =
        (fmap(f) _ compose duplicate)(wa)
}
```

`extend` と `duplicate` のデフォルト実装が提供されているため、どちらか一方をオーバーライドするだけで済みます。

これらの関数の背後にある直感は、一般的に余モナドを `a` 型の値で満たされたコンテナと考えることに基づいています（積型余モナドはちょうど一つの値の特殊なケースでした）。`easily accessible through extract` という「現在の」値があります。余 Kleisli 射は現在の値に焦点を当てた計算を実行しますが、周囲のすべての値にアクセスできます。Conway のライフゲームを考えてみてください。各セルには値が含まれています（通常は単に `True` または `False`）。ライフゲームの余モナドは「現在の」セルに焦点を当てたセルのグリッドに相当します。

それでは `duplicate` は何をしますか？ それは余モナドのコンテナ `w a` を取り、コンテナのコンテナ `w (w a)` を生成します。各コンテナが `w a` 内の異なる `a` に焦点を当てているという考えです。ライフゲームであれば、外側のグリッドの各セルに、異なるセルに焦点を当てた内側のグリッドが含まれます。

そして `extend` を見てください。それは余 Kleisli 射と `a` の `w a` で満たされた余モナドのコンテナを取ります。それはそれらのすべての `a` に計算を適用し、それらを `b` に置き換えます。結果は `b` で満たされた余モナドのコンテナです。`extend` は一つ一つの `a` に焦点を移動させ、それらのそれぞれに余 Kleisli 射を適用することでこれを行います。ライフゲームでは、余 Kleisli 射は現在のセルの新しい状態を計算します。そのためには、そのコンテキスト – おそらく最も近い隣人 – を見ることになります。`extend` のデフォルト実装はこのプロセスを示し

ています。最初に `duplicate` を呼び出してすべての可能な焦点を生成し、それからそれぞれに `f` を適用します。

23.4 ストリーム余モナド

コンテナの一つの要素から別の要素に焦点を移動するこのプロセスは、無限ストリームの例で最もよく示されます。そのようなストリームはリストのようなものですが、空のコンストラクタがありません：

```
data Stream a = Cons a (Stream a)

case class Stream[A](h: () => A, t: () => Stream[A])
```

それは自明に `Functor` です：

```
instance Functor Stream where
    fmap f (Cons a as) = Cons (f a) (fmap f as)

implicit val streamFunctor = new Functor[Stream] {
    def fmap[A, B](f: A => B)(fa: Stream[A]): Stream[B] = fa match {
        case Stream(a, as) =>
            Stream(() => f(a()), () => fmap(f)(as())))
    }
}
```

ストリームの焦点はその最初の要素なので、こちらが `extract` の実装です:

```
extract (Cons a _) = a

def extract[A](wa: Stream[A]): A = wa match {
  case Stream(a, _) => a()
}
```

`duplicate` は異なる要素に焦点を当てたストリームのストリームを生成します。

```
duplicate (Cons a as) = Cons (Cons a as) (duplicate as)

def duplicateS[A](wa: Stream[A]): Stream[Stream[A]] = wa match {
  case s @ Stream(a, as) =>
    Stream(() => s, () => duplicateS(as()))
}
```

最初の要素は元のストリームで、二番目の要素は元のストリームの尾です。三番目の要素はその尾で、以下同様に無限に続きます。

こちらが完全なインスタンスです:

```
instance Comonad Stream where
  extract (Cons a _) = a
  duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

```
implicit val streamComonad = new Comonad[Stream] {  
    def extract[A](wa: Stream[A]): A =  
        wa match {  
            case Stream(a, _) => a()  
        }  
  
    def duplicate[A](wa: Stream[A]): Stream[Stream[A]] = wa match {  
        case s @ Stream(a, as) =>  
            Stream(() => s, () => duplicate(as()))  
    }  
}
```

これはストリームを見る非常に機能的な方法です。命令型言語では、ストリームを一つの位置にシフトする `advance` メソッドから始めるかもしれません。ここでは、`duplicate` は一気にすべてのシフトされたストリームを生成します。Haskell の遅延評価はこれを可能にし、さらに望ましいものにします。もちろん、`Stream` を実用的にするために、`advance` の類似物も実装します:

```
tail :: Stream a -> Stream a  
tail (Cons a as) = as  
  
def tail[A]: Stream[A] => Stream[A] = {  
    case Stream(a, as) => as()  
}
```

しかし、それは余モナドインターフェースの一部ではありません。

デジタル信号処理に何らかの経験があるなら、ストリームの余 Kleisli 射はただのデジタルフィルタであることがすぐにわかります。そして、`extend` はフィルタリングされたストリームを生成します。

簡単な例として、移動平均フィルタを実装しましょう。こちらはストリームの最初の n 要素を合計する関数です:

```
sumS :: Num a => Int -> Stream a -> a
sumS n (Cons a as) = if n <= 0 then 0 else a + sumS (n - 1) as

def sumS[A](n: Int)(stm: Stream[A])(implicit numeric: Numeric[A]): A =
  stm match {
    case Stream(a, as) =>
      import numeric._
      if (n <= 0) zero else a() + sumS(n - 1)(as())
  }
```

こちらはストリームの最初の n 要素の平均を計算する関数です:

```
average :: Fractional a => Int -> Stream a -> a
average n stm = (sumS n stm) / (fromIntegral n)

def average[A](n: Int)
  (implicit fractional: Fractional[A]): Stream[A] => A = stm => {
  import fractional._
  sumS(n)(stm) / fromInt(n)
```

```
}
```

部分適用された `average n` は余 Kleisli 射なので、全体のストリームに `extend` することができます:

```
movingAvg :: Fractional a => Int -> Stream a -> Stream a
movingAvg n = extend (average n)
```

```
def movingAvg[A](n: Int)(stm: Stream[A])
  (implicit fractional: Fractional[A]): Stream[A] =
  streamComonad.
    extend(average(n)(fractional))(stm)
```

結果は移動平均のストリームです。

ストリームは一方向、一次元の余モナドの例です。それは簡単に双方向にしたり、二次元以上に拡張することができます。

23.5 圏論的に余モナドを定義する

圏論で余モナドを定義することは、双対性を利用することで直截的な演習です。モナドと同様に、私たちは自己関手 T から始めます。モナドを定義する 2 つの自然変換 η と μ は、余モナドに対して単純に逆転します:

$$\varepsilon :: T \rightarrow I$$

$$\delta :: T \rightarrow T^2$$

これらの変換のコンポーネントは **extract** と **duplicate** に対応します。余モナド則はモナド則の鏡像です。ここに大きな驚きはありません。

次に、随伴からモナドを導出します。双対性は随伴を逆転させます：左随伴は右随伴になり、その逆もまた然りです。そして、 $R \circ L$ の合成がモナドを定義するので、 $L \circ R$ は余モナドを定義するはずです。随伴の余単位：

$$\varepsilon :: L \circ R \rightarrow I$$

は、実際には余モナドの定義で見ると同じ ε です—または、Haskell の **extract** としてのコンポーネントとしてです。随伴の単位：

$$\eta :: I \rightarrow R \circ L$$

を使用して、 $L \circ R$ の中間に $R \circ L$ を挿入し、 $L \circ R \circ L \circ R$ を生成することができます。 T から T^2 を作ることが δ を定義し、それが余モナドの定義を完成させます。

また、モナドがモノイドであることも見てきました。この言明の双対は、余モノイドの使用を必要とするので、何が余モノイドですか？单一対象圏としてのモノイドの元の定義は、何か興味深いものに双対化されません。すべての自己射の方向を逆にしても、別のモノイドが得られます。しかし、モナドへの私たちのアプローチで、私たちはモノイダル圏における対象としてのモノイドのより一般的な定義を使用

しました。その構造は二つの射に基づいていました:

$$\mu : m \otimes m \rightarrow m$$

$$\eta : i \rightarrow m$$

これらの射の逆転はモノイダル圏内の余モノイドを生成します:

$$\delta : m \rightarrow m \otimes m$$

$$\varepsilon : m \rightarrow i$$

Haskell で余モノイドの定義を書くことができます:

```
class Comonoid m where
    split :: m -> (m, m)
    destroy :: m -> ()  
  
trait Comonoid[M] {
    def split(x: M): (M, M)
    def destroy(x: M): Unit
}
```

しかし、それはかなり自明です。明らかに `destroy` はその引数を無視します。

```
destroy _ = ()  
  
def destroy(x: M): Unit = ()
```

`split` は単なる関数のペアです:

```
split x = (f x, g x)
```

```
def split(x: M): (M, M) = (f(x), g(x))
```

次に、モノイドの単位則に双対する余モノイド則を考えます。

```
lambda . bimap destroy id . split = id
rho . bimap id destroy . split = id
```

```
(lambda compose bimap(destroy)(identity[M]))
.compose(split) == identity[M]
```

```
(rho compose bimap(identity[M])(destroy))
.compose(split) == identity[M]
```

ここで、`lambda` と `rho` はそれぞれ左単位子と右単位子です（モノイダル圏の定義を参照してください）。定義を差し込むと、`g = id` を得ます。同様に、二番目の規則は `f = id` に展開されます。結論として：

```
split x = (x, x)
```

```
def split(x: M): (M, M) = (x, x)
```

これは Haskell (および、一般的には圏 `Set`) では、すべての対象が自明な余モノイドであることを示しています。

幸いなことに、余モノイドを定義するための他のより興味深いモノイダル圏があります。その一つは自己関手の圏です。そして、モナドが自己関手の圏のモノイドであるように、

余モナドは自己関手の圏の余モノイドです。

23.6 ストア余モナド

もう一つの重要な余モナドの例は、状態モナドの双対です。それは余状態余モナドまたは、代替的にストア余モナドと呼ばれます。

以前に、状態モナドは指數関数を定義する随伴によって生成されることを見ました:

$$Lz = z \times s$$

$$Ra = s \Rightarrow a$$

私たちは同じ随伴を使って余状態余モナドを定義します。余モナドは $L \circ R$ の合成によって定義されます:

$$L(Ra) = (s \Rightarrow a) \times s$$

これを Haskell に翻訳すると、左側に **Product** 関手、右側に **Reader** 関手を持つ随伴から始めます。**Product** の後に **Reader** を組み合わせることは、次の定義と同等です:

```
data Store s a = Store (s -> a) s

case class Store[S, A](run: S => A, s: S)
```

随伴の余単位を対象 a で取ると、射は以下のようにになります:

$$\varepsilon_a :: ((s \Rightarrow a) \times s) \rightarrow a$$

または、Haskell の表記で:

```
counit (Prod (Reader f) s)) = f s

def counit[S, A](a: Product[S, Reader[S, A]]): A = a match {
  case Product((Reader(f), s)) => f(s)
}
```

これが私たちの `extract` になります:

```
extract (Store f s) = f s

def extract[A](wa: Store[S, A]): A = wa match {
  case Store(f, s) => f(s)
}
```

随伴の単位:

```
unit :: a -> Reader s (Product a s)
unit a = Reader (\s -> Prod a s)

def unit[S, A](a: A): Reader[S, Product[S, A]] =
  Reader(s => Product((a, s)))
```

は部分的に適用されたデータコンストラクタとして書き直すことができます:

```
Store f :: s -> Store f s

object Store {
  def apply[S, A](run: S => A): S => Store[S, A] =
    s => new Store(run, s)
}
```

δ 、または `duplicate` を、水平合成として構成します:

$$\begin{aligned}\delta &:: L \circ R \rightarrow L \circ R \circ L \circ R \\ \delta &= L \circ \eta \circ R\end{aligned}$$

最左端の L を通して η をこっそりと通す必要があります。それは `Product` 関手であるため、 η 、または `Store f` をペアの左側要素に作用させることを意味します (それが `Product` に対する `fmap` が行うことです)。私たちは得ます:

```
duplicate (Store f s) = Store (Store f) s

def duplicate[A](wa: Store[S, A]): Store[S, Store[S, A]] = wa match {
  case Store(f, s) => Store(Store(f), s)
}
```

(δ の式で、 L と R は恒等自然変換を表し、そのコンポーネントは恒等射です。)

こちらが `Store` 余モナドの完全な定義です:

```
instance Comonad (Store s) where
  extract (Store f s) = f s
  duplicate (Store f s) = Store (Store f) s

implicit def storeComonad[S] = new Comonad[Store[S, ?]] {
  def extract[A](wa: Store[S, A]): A = wa match {
    case Store(f, s) => f(s)
  }

  override def duplicate[A](wa: Store[S, A]): Store[S, Store[S, A]] =
    wa match {
      case Store(f, s) => Store(Store(f), s)
    }
}
```

あなたは **Reader** 部分の **Store** を、**a** の一般化されたコンテナとして考えるかもしれません。それは **s** 型の要素を使ってキー付けされています。例えば、**s** が **Int** であれば、**Reader Int a** は **a** の無限双方向ストリームです。**Store** はこのコンテナをキー型の値と組み合わせます。この場合、**extract** はこの整数を使用して無限ストリームをインデックス化します。第二のコンポーネントの **Store** を現在の位置と考えることができます。

この例を続けると、**duplicate** は **Int** でインデックス付けされた新しい無限ストリームを作成します。このストリームにはストリームが要素として含まれます。特に、現在の位置では、元のストリームが含まれます。しかし、他の **Int** (正または負) をキーとして使用した場合、その新しいインデックスに位置するシフトされたストリームを取得します。

一般的に、**extract** が **duplicate** された **Store** に作用するとき、それは元の **Store** を生成します (実際には、余モナドの恒等則は **extract . duplicate = id** を述べています)。

Store 余モナドは、**Lens** ライブラリの理論的基礎として重要な役割を果たします。概念的には、**Store s a** 余モナドは、型 **s** をインデックスとして使用してデータ型 **a** の特定の部分構造に「焦点を当てる」(レンズのように) ことのアイデアを包含します。特に、型が:

a -> Store s a

```
A => Store[S, A]
```

の関数は、次のペアの関数と等価です:

```
set :: a -> s -> a  
get :: a -> s
```

```
def set[A, S]: A => S => A  
def get[A, S]: A => S
```

`a` が積型の場合、`set` は `a` 内の `s` 型のフィールドを設定しながら、`a` の変更されたバージョンを返すとして実装することができます。同様に、`get` は `a` から `s` フィールドの値を読み取るために実装することができます。次のセクションでこれらのアイデアをさらに探求します。

23.7 チャレンジ

1. **Store** 余モナドを使用して Conway のライフゲームを実装してください。ヒント: `s` にどんな型を選びますか？

24

F-代数

私 たちはモノイドのいくつかの形式を見てきました: 集合として、单一対象圏として、モノイダル圏の対象として。このシンプルな概念から、どれだけ多くの知見を引き出すことができるでしょうか？試してみましょう。集合 m と一対の関数を用いたモノイドの定義を考えます:

$$\begin{aligned}\mu &:: m \times m \rightarrow m \\ \eta &:: 1 \rightarrow m\end{aligned}$$

ここで、 1 は **Set** 内の終対象 – つまり単集合です。最初の関数は乗算を定義しています (要素の対を取り、その積を返します)、二番目の関数は m から単位要素を選びます。この二つの関数の任意の選択がモノ

イドになるわけではありません。それには追加の条件、結合則と単位則が必要です。しかし、今はそれを忘れて「潜在的なモノイド」だけを考えてみましょう。関数の対は、二つの関数集合のデカルト積の要素です。これらの集合は指標対象として表現できることが知られています:

$$\begin{aligned}\mu &\in m^{m \times m} \\ \eta &\in m^1\end{aligned}$$

これら二つの集合のデカルト積は:

$$m^{m \times m} \times m^1$$

高校の代数を使うと(これはデカルト閉圏でも機能します)、次のように書き換えられます:

$$m^{m \times m + 1}$$

+記号は **Set** 内の余積を示します。我々はただ関数の対を单一の関数に置き換えただけです — 集合の要素:

$$m \times m + 1 \rightarrow m$$

この関数集合の任意の要素は潜在的なモノイドです。

この定式化の美しさは、それが興味深い一般化につながることです。例えば、この言語を使って群をどのように記述しますか？群は各要素に逆要素を割り当てる追加の関数を持つモノイドです。後者は $m \rightarrow m$ という型の関数です。例えば、整数は加算を二項演算とし、ゼロを単

位要素とし、否定を逆要素として群を形成します。群を定義するためには、関数の三つ組から始めます:

$$m \times m \rightarrow m$$

$$m \rightarrow m$$

$$1 \rightarrow m$$

以前と同様に、これらの三つ組を一つの関数集合に結合できます:

$$m \times m + m + 1 \rightarrow m$$

私たちは一つの二項演算子(加算)、一つの単項演算子(否定)、一つの零項演算子(恒等要素 – ここではゼロ)で始めました。それらを一つの関数に結合しました。この署名を持つすべての関数は潜在的な群を定義します。

我々はこのように続けることができます。例えば、環を定義するためには、もう一つの二項演算子と一つの零項演算子を加えます、等々。毎回、我々は左辺がべきの和(零次べきを含む可能性がある – 終対象)であり、右辺が集合自体である関数型に終わります。

今、我々は一般化に狂うことができます。まず、集合を対象に置き換え、関数を射に置き換えることができます。我々は n 項演算子を n 項積からの射として定義できます。それは我々が有限積をサポートする圏を必要とすることを意味します。零項演算子については、終対象の存在を要求します。従って、我々はデカルト圏が必要です。これらの演算子を結合するためには、指数対象が必要です、それがデカルト

閉圏です。最後に、我々の代数的ないたずらを完成させるために余積が必要です。

あるいは、我々は我々が導出した方法を忘れて、最後の積に集中することができます。射の左辺に関する積の和は自己関手を定義します。任意の自己関手 F を選んだらどうでしょうか？ その場合、我々は我々の圏に何の制約も課さなくてよいです。我々が得るものは F -代数と呼ばれます。

F -代数は自己関手 F 、対象 a 、そして射

$$Fa \rightarrow a$$

からなる三つ組です。対象はしばしば台となる対象または、プログラミングの文脈では台となる型と呼ばれます。射はしばしば評価関数または構造マップと呼ばれます。関手 F は表現を形成し、射はそれらを評価すると考えてください。

こちらが F -代数の Haskell 定義です:

```
type Algebra f a = f a -> a
```

```
type Algebra[F[_], A] = F[A] => A
```

これは代数をその評価関数と同一視します。

モノイドの例で、問題の関手は:

```
data MonF a = MEmpty | MAppend a a

sealed trait MonF[+A]
case object MEmpty extends MonF[Nothing]
case class MAppend[A](m: A, n: A) extends MonF[A]
```

これは Haskell で $1 + a \times a$ です (代数的データ構造を思い出してください)。

環は以下の関手を使用して定義されます:

```
data RingF a = RZero
  | ROne
  | RAdd a a
  | RMul a a
  | RNeg a

sealed trait RingF[+A]
case object RZero extends RingF[Nothing]
case object ROne extends RingF[Nothing]
case class RAdd[A](m: A, n: A) extends RingF[A]
case class RMul[A](m: A, n: A) extends RingF[A]
case class RNeg[A](n: A) extends RingF[A]
```

これは Haskell で $1 + 1 + a \times a + a \times a + a$ です。

整数の集合の例は環です。我々は台となる型として **Integer** を選び、評価関数を定義することができます:

```
evalZ :: Algebra RingF Integer
evalZ RZero      = 0
evalZ ROne       = 1
evalZ (RAdd m n) = m + n
evalZ (RMul m n) = m * n
evalZ (RNeg n)   = -n
```

```
def evalZ: Algebra[RingF, Int] = {
    case RZero => 0
    case ROne => 1
    case RAdd(m, n) => m + n
    case RMul(m, n) => m * n
    case RNeg(n) => -n
}
```

同じ関手 `RingF` に基づくより多くの F-代数があります。例えば、多項式は環を形成し、正方行列もそうです。

関手の役割は、代数の評価者を使用して評価できる表現を生成することです。これまでに、我々は非常に単純な表現しか見ていません。我々はしばしば、再帰を使用して定義されるより複雑な表現に関心があります。

24.1 再帰

任意の表現ツリーを生成する一つの方法は、関手定義内の変数 `a` を再帰的に置き換えることです。例えば、環内の任意の表現はこのツリー様のデータ構造によって生成されます:

```
data Expr = RZero
  | ROne
  | RAdd Expr Expr
  | RMul Expr Expr
  | RNeg Expr

sealed trait Expr
case object RZero extends Expr
case object ROne extends Expr
case class RAdd(e1: Expr, e2: Expr) extends Expr
case class RMul(e1: Expr, e2: Expr) extends Expr
case class RNeg(e: Expr) extends Expr
```

我々は元の環評価者をその再帰バージョンで置き換えることができます:

```
evalZ :: Expr -> Integer
evalZ RZero      = 0
evalZ ROne       = 1
evalZ (RAdd e1 e2) = evalZ e1 + evalZ e2
evalZ (RMul e1 e2) = evalZ e1 * evalZ e2
```

```
evalZ (RNeg e)      = -(evalZ e)

def evalZ: Expr => Int = {
    case RZero => 0
    case ROne => 1
    case RAdd(e1, e2) => evalZ(e1) + evalZ(e2)
    case RMul(e1, e2) => evalZ(e1) * evalZ(e2)
    case RNeg(e) => -evalZ(e)
}
```

これはまだ非常に実用的ではありません、なぜなら我々はすべての整数を一の和として表現する必要があるからです、しかし一つの方法としては機能します。

しかし、我々はF-代数を使用して表現ツリーをどのように記述することができますか？ 我々は、関手の定義における自由型変数を、置換の結果で再帰的に置き換えるプロセスを何らかの形で形式化する必要があります。このプロセスを段階的に想像してみてください。まず、次のような深さ1のツリーを定義します：

```
type RingF1 a = RingF (RingF a)

type RingF1[A] = RingF[RingF[A]]
```

我々は `RingF a` によって生成される深さゼロのツリーで `RingF` の穴を埋めています。深さ2のツリーは同様に得られます：

```
type RingF2 a = RingF (RingF (RingF a))
```

```
type RingF2[A] = RingF[RingF[RingF[A]]]
```

我々はまた次のように書くこともできます:

```
type RingF2 a = RingF (RingF1 a)
```

```
type RingF2[A] = RingF[RingF1[A]]
```

このプロセスを続けると、我々は次のような記号的な方程式を書くことができます:

```
type RingFn+1 a = RingF (RingFn a)
```

概念的には、このプロセスを無限回繰り返すと、最終的には **Expr** にたどり着きます。注目すべきは、**Expr** は **a** に依存しないことです。我々の旅の出発点は何であれ、我々は常に同じ場所に終わります。これは任意の自己関手について任意の圏で常に真であるわけではありませんが、圏 **Set** ではうまくいきます。

もちろん、これは手振りの議論ですが、後でそれをもっと厳密にするつもりです。

自己関手を無限回適用すると、**不動点**が生成されます。これは次のように定義される対象です:

$$\text{Fix } f = f(\text{Fix } f)$$

この定義の直観は、 $\text{Fix } f$ を得るために無限回 f を適用したので、もう一度適用しても何も変わらないということです。Haskellでの不動点の定義は：

```
newtype Fix f = Fix (f (Fix f))

case class Fix[F[_]](x: F[Fix[F]])
```

おそらく、コンストラクタの名前が定義されている型の名前と異なる場合、より読みやすいでしょう、例えば：

```
newtype Fix f = In (f (Fix f))

sealed trait Fix[F[_]]
case class In[F[_]](f: F[Fix[F]])
```

しかし、私は受け入れられた表記に固執します。コンストラクタ `Fix` (あるいは、好みであれば `In`) は関数と見なすことができます：

```
Fix :: f (Fix f) -> Fix f

object Fix {
    def apply[F[_]](f: F[Fix[F]]): Fix[F] = new Fix(f)
}
```

関数があり、一つのレベルの関手適用を剥がすこともあります：

```

unFix :: Fix f -> f (Fix f)
unFix (Fix x) = x

def unFix[F[_]]: Fix[F] => F[Fix[F]] = {
  case Fix(x) => x
}

```

これらの二つの関数は互いに逆です。後でこれらの関数を使用します。

24.2 F-代数の圏

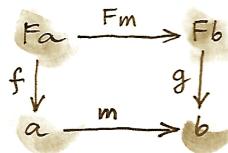
ここには本の中で最も古いトリックがあります: 新しいオブジェクトの構成法を考えたら、それらが圏を形成するかどうかを見てください。驚くべきことではありませんが、与えられた自己関手 F に対する代数は圏を形成します。その圏の対象は代数 – 元の圏 \mathbf{C} からの対象 a と射 $Fa \rightarrow a$ からなるペアです。

この図を完成させるためには、F-代数の圏における射を定義する必要があります。射は一つの代数 (a, f) から別の代数 (b, g) へのマッピングでなければなりません。それを元の圏の対象から対象への射 m として定義します。どんな射でも良いわけではありません: 我々はそれが二つの評価者と互換性を持つことを望みます。(このような構造を保存する射を**準同型**と呼びます。) F-代数の準同型を定義する方法は次のとおりです。まず、 m を次のマッピングにリフトすることができます:

$$Fm :: Fa \rightarrow Fb$$

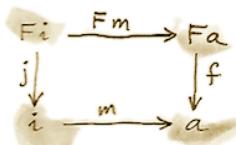
それに続いて b への g を使用します。または、 Fa から a への f を使用し、それに続いて m を適用することもできます。二つのパスを等しいものとしたいです:

$$g \circ Fm = m \circ f$$



これが実際に圏であることを自分自身に確認するのは簡単です(ヒント: C からの恒等射がうまく機能し、準同型の合成は準同型です)。

F -代数の圏における始対象が存在する場合、それは初期代数と呼ばれます。この初期代数の台を i 、その評価者を $j :: Fi \rightarrow i$ と呼びましょう。実は j 、初期代数の評価者は同型です。これは Lambek の定理として知られています。証明は始対象の定義に依存します。それは任意の他の F -代数への唯一の準同型 m が存在することを要求します。 m が準同型であるので、以下の図式が可換でなければなりません:



次に、台が F_i であるような代数を構成しましょう。そのような代数の評価者は $F(F_i)$ から F_i への射でなければなりません。我々は簡単に j をリフトすることでそのような評価者を構成することができます:

$$Fj :: F(F_i) \rightarrow F_i$$

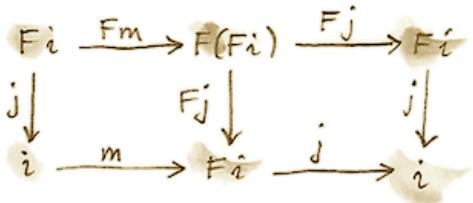
(i, j) が初期代数であるので、 (F_i, F_j) に対する唯一の準同型 m が存在しなければなりません。以下の図式が可換でなければなりません:

$$\begin{array}{ccc} F_i & \xrightarrow{Fm} & F(F_i) \\ j \downarrow & & \downarrow F_j \\ i & \xrightarrow{m} & F_i \end{array}$$

しかし、我々はまたこの自明に可換な図式も持っています(両方のパスが同じです!):

$$\begin{array}{ccc} F(F_i) & \xrightarrow{Fj} & F_i \\ F_j \downarrow & & \downarrow j \\ F_i & \xrightarrow{j} & i \end{array}$$

これを、 (F_i, F_j) から (i, j) への代数の準同型として j をマッピングするものとして解釈することができます。我々はこれら二つの図式を一緒に接着して、次のように得ることができます:



この図式は、 $j \circ m$ が代数の準同型であることを示していると解釈することができます。この場合、二つの代数は同じです。さらに、 (i, j) が初期であるので、それ自体に対しては唯一の準同型が存在し、それが恒等射 \mathbf{id}_i です – これは代数の準同型です。したがって $j \circ m = \mathbf{id}_i$ です。この事実と左図の可換性を使用して、 $m \circ j = \mathbf{id}_{F_i}$ を証明することができます。これは m が j の逆であり、したがって j は F_i と i の間の同型です:

$$F_i \cong i$$

しかし、それはただ i が F の不動点であると言っているだけです。それが元の手振りの議論の背後にある公式的な証明です。

Haskell に戻ると: 我々は i を我々の `Fix f`、 j を我々のコンストラクタ `Fix`、その逆を `unFix` と認識します。Lambek の定理の同型は、初期代数を得るために関手 f を取り、その引数 a を `Fix f` に置き換えると教えてくれます。また、不動点が a に依存しない理由もわかります。

24.3 自然数

自然数も F-代数として定義できます。出発点は次の二つの射です:

$$\text{zero} :: 1 \rightarrow N$$

$$\text{succ} :: N \rightarrow N$$

最初の射はゼロを選び、二番目の射は全ての数をその後者に写像します。以前と同様に、これら二つを一つに組み合わせることができます:

$$1 + N \rightarrow N$$

左辺は Haskell で次のように書くことができる関手を定義します:

```
data NatF a = ZeroF | SuccF a

sealed trait NatF[+A]
case object ZeroF extends NatF[Nothing]
case class SuccF[A](a: A) extends NatF[A]
```

この関手の不動点 (それが生成する初期代数) は Haskell で次のようにエンコードできます:

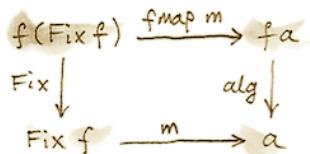
```
data Nat = Zero | Succ Nat

sealed trait Nat
case object Zero extends Nat
case class Succ(n: Nat) extends Nat
```

自然数はゼロか他の数の後者のどちらかです。これは自然数のペアノ表現として知られています。

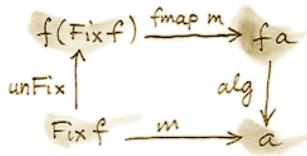
24.4 カタモルフィズム

初期性条件を Haskell の記法を使って書き直しましょう。初期代数を `Fix f` と呼び、その評価者をコンストラクタ `Fix` とします。初期代数から同じ関手上の任意の他の代数への唯一の射 `m` が存在します。キャリアが `a` で評価者が `alg` である代数を選びましょう。



ところで、`m` とは何かに注意してください: それは不動点の評価者、すなわち全体の再帰的表現ツリーの評価者です。それを実装する一般的な方法を見つけましょう。

Lambek の定理は、コンストラクタ `Fix` が同型であることを教えてくれます。私たちはその逆を `unFix` と呼びました。したがって、この図の一つの射を反転させることができます:



この図の交通条件を書き下しましょう:

$$m = alg . fmap m . unFix$$

この方程式を m の再帰的定義として解釈することができます。この再帰は関手 f を使用して構成された任意の有限ツリーに対して終了する必要があります。それは $fmap m$ が関手 f の最上層の下で動作することに気づくことでわかります。言い換えれば、それは元のツリーの子に作用します。子は常に元のツリーより一つレベルが浅いです。

m を $\text{Fix } f$ を使用して構成されたツリーに適用すると、次のようなことが起こります。 $unFix$ の作用はコンストラクタを剥がし、ツリーの最上層を露出させます。それから、私たちはトップノードの全ての子に m を適用します。これは a 型の結果を生成します。最後に、非再帰的評価者 alg を適用することによってこれらの結果を結合します。重要な点は、私たちの評価者 alg が単純な非再帰的関数であることです。

任意の代数 alg に対してこれを行うことができるので、代数をパラメータとして取り、私たちが m と呼んだ関数を与える高階関数を定義

することは理にかなっています。この高階関数はカタモルフィズムと呼ばれます:

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix

def cata[F[_], A](alg: F[A] => A)
  (implicit F: Functor[F]): Fix[F] => A =
  alg.compose(F.fmap(cata(alg)) _ compose unFix)
```

それがどのように機能するかの例を見てみましょう。自然数を定義する関手を取ります:

```
data NatF a = ZeroF | SuccF a

sealed trait NatF[+A]
case object ZeroF extends NatF[Nothing]
case class SuccF[A](a: A) extends NatF[A]
```

キャリア型として (Int, Int) を選び、私たちの代数を次のように定義しましょう:

```
fib :: NatF (Int, Int) -> (Int, Int)
fib ZeroF = (1, 1)
fib (SuccF (m, n)) = (n, m + n)
```

```
def fib: NatF[(Int, Int)] => (Int, Int) = {
    case ZeroF => (1, 1)
    case SuccF((m, n)) => (n, m + n)
}
```

簡単に自分自身を納得させることができるでしょう、カタモルフィズム `cata` `fib` はフィボナッチ数を計算します。

一般的に、`NatF` の代数は、現在の要素の値を前の要素の値の観点から定義する漸化式を定義します。カタモルフィズムはそのシーケンスの `n` 番目の要素を評価します。

24.5 置み込み

`e` のリストは以下の関手の初期代数です:

```
data ListF[e] = NilF | ConsF[e] a

sealed trait ListF[+E, +A]
case object NilF extends ListF[Nothing, Nothing]
case class ConsF[E, A](h: E, t: A) extends ListF[E, A]
```

実際、変数 `a` を再帰の結果で置き換えると、次のようにになります:

```
data List[e] = Nil | Cons[e] (List[e])
```

```
sealed trait List[+E]
case object Nil extends List[Nothing]
case class Cons[E](h: E, t: List[E]) extends List[E]
```

リスト関手の代数は、特定のキャリア型を選び、二つのコンストラクタに対するパターンマッチングを行う関数を定義します。その `NilF` の値は空リストを評価する方法を教えてくれ、その `ConsF` の値は現在の要素を以前に蓄積された値と組み合わせる方法を教えてくれます。

例えば、リストの長さを計算するために使用できる代数は次のとおりです(キャリア型は `Int` です)：

```
lenAlg :: ListF[e] Int -> Int
lenAlg (ConsF e n) = n + 1
lenAlg NilF = 0

def lenAlg[E]: ListF[E, Int] => Int = {
  case ConsF(e, n) => n + 1
  case NilF => 0
}
```

実際、結果のカタモルフィズム `cata` `lenAlg` はリストの長さを計算します。評価者は(1)リスト要素と累積器を取り、新しい累積器を返す関数と(2)ここではゼロである開始値の組み合わせです。値の型と累積器の型はキャリア型によって与えられます。

これを伝統的な Haskell の定義と比較してみてください：

```
length = foldr (\e n -> n + 1) 0

def length[E](l: List[E]): Int =
  l.foldRight(0)((e, n) => n + 1)
```

`foldr`への二つの引数はまさに代数の二つのコンポーネントです。
別の例を試してみましょう:

```
sumAlg :: ListF[Double, Double] -> Double
sumAlg (ConsF e s) = e + s
sumAlg NilF = 0.0

def sumAlg: ListF[Double, Double] => Double = {
  case ConsF(e, s) => e + s
  case NilF => 0.0
}
```

再び、これを比較してください:

```
sum = foldr (\e s -> e + s) 0.0

def sum(l: List[Double]): Double =
  l.foldRight(0.0)((e, s) => e + s)
```

ご覧のとおり、`foldr`はリストに対するカタモルフィズムの便利な特
殊化に過ぎません。

24.6 余代数

通常どおり、我々は F -余代数の双対構造を持っています。ここでは射の方向が逆転します:

$$a \rightarrow Fa$$

与えられた関手の余代数も圏を形成し、余代数構造を保つ準同型があります。その圏の終対象 (t, u) は終(または最終)余代数と呼ばれます。他の代数 (a, f) に対しては、次の図式を可換にする唯一の準同型 m が存在します:

$$\begin{array}{ccc} Ft & \xleftarrow{Fm} & Fa \\ u \uparrow & & f \uparrow \\ t & \xleftarrow{m} & a \end{array}$$

終余代数は、射 $u :: t \rightarrow Ft$ が同型であるという意味で関手の不動点です(余代数のための Lambek の定理):

$$Ft \cong t$$

終余代数は通常、(無限に) データ構造や遷移システムを生成するレシピとしてプログラミングで解釈されます。

カタモルフィズムが初期代数を評価するために使用されるように、アナモルフィズムは終余代数を共評価するために使用されます:

```
ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg

def ana[F[_], A](coalg: A => F[A])
    (implicit F: Functor[F]): A => Fix[F] =
(Fix.apply[F] _).compose(F.fmap(ana(coalg)) _ compose coalg)
```

余代数の標準的な例は、その不動点が型 `e` の要素の無限ストリームである関手に基づいています。これがその関手です:

```
data StreamF e a = StreamF e a
    deriving Functor

case class StreamF[E, A](h: E, t: A)

implicit def streamFFunctor[E] = new Functor[StreamF[E, ?]] {
    def fmap[A, B](f: A => B)(fa: StreamF[E, A]): StreamF[E, B] =
    ...
}
```

そして、これがその不動点です:

```
data Stream e = Stream e (Stream e)

case class Stream[E](h: E, t: Stream[E])
```

`StreamF` の余代数は、型 `a` の種を取り、要素と次の種からなるペア (`StreamF` はペアのファンシーな名前です) を生成する関数です。

簡単な余代数の例は、平方数のリストや逆数のリストを生成するものを簡単に生成できます。

より興味深い例は、素数のリストを生成する余代数です。トリックは無限リストをキャリアとして使用することです。私たちの開始種はリスト `[2..]` です。次の種はこのリストの尾であり、すべての 2 の倍数が除去されます。それは 3 から始まる奇数のリストです。次のステップでは、このリストの尾を取り、すべての 3 の倍数を除去します、そして以下同様です。あなたはエラトステネスの篩の始まりを認識するかもしれません。この余代数は次の関数によって実装されます:

```
era :: [Int] -> StreamF Int [Int]
era (p : ns) = StreamF p (filter (notdiv p) ns)
  where notdiv p n = n `mod` p /= 0

def era: List[Int] => StreamF[Int, List[Int]] = {
  case p :: ns =>
    def notdiv(p: Int)(n: Int): Boolean =
      n % p != 0
    StreamF(p, ns.filter(notdiv(p)))
}
```

この余代数のアナモルフィズムは素数のリストを生成します:

```
primes = ana era [2..]

// just imagine that the list is infinite
def primes =
  ana(era)(streamFFunctor)((1 to 10).toList)
```

ストリームは無限リストなので、それを Haskell リストに変換することが可能ですが。それを行うために、私たちは同じ関手 `StreamF` を使用して代数を形成し、それにカタモルフィズムを走らせることができます。例えば、これはストリームをリストに変換するカタモルフィズムです:

```
toListC :: Fix (StreamF e) -> [e]
toListC = cata al
  where al :: StreamF e [e] -> [e]
        al (StreamF e a) = e : a

def toListC[E]: Fix[StreamF[E, ?]] => List[E] = {
  def al: StreamF[E, List[E]] => List[E] = {
    case StreamF(e, a) => e :: a
  }
  cata[StreamF[E, ?], List[E]](al)
}
```

ここでは、同じ不動点が同じ自己関手の初期代数と終余代数の両方で同時に使用されています。任意の圏では必ずしもこのようになるわけ

ではありません。一般に、自己関手は多くの（または無い）不動点を持つことがあります。初期代数はいわゆる最小不動点であり、終余代数は最大不動点です。しかし、Haskell では両方とも同じ式で定義されており、一致します。

リストのためのアナモルフィズムは unfold と呼ばれます。有限リストを作成するためには、関手は **Maybe** ペアを生成するように変更されます：

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

```
def unfoldr[A, B]: (B => Option[(A, B)]) => B => List[A]
```

Nothing の値はリストの生成を終了します。

レンズに関連する余代数の興味深いケースがあります。レンズは取得関数と設定関数のペアとして表されることがあります：

```
set :: a -> s -> a  
get :: a -> s
```

```
def set[A, S]: A => S => A  
def get[A, S]: A => S
```

ここで、**a** は通常、型 **s** のフィールドを持ついくつかの積データ型です。取得関数はそのフィールドの値を取得し、設定関数はそのフィー

ルドを新しい値で置き換えます。これら二つの関数は一つに結合することができます:

```
a -> (s, s -> a)
```

```
A => ((S, S => A))
```

この関数をさらに次のように書き換えることができます:

```
a -> Store s a
```

```
A => Store[S, A]
```

ここで、関手を定義しました:

```
data Store s a = Store (s -> a) s
```

```
case class Store[S, A](run: S => A, s: S)
```

注意してください、これは積の和から構成される単純な代数的関手ではありません。それは指数 a^s を含んでいます。

レンズはこの関手の余代数であり、キャリア型は **a** です。前に見たように、**Store s** はまた余モナドでもあります。うまく動作するレンズは、余モナド構造と互換性のある余代数に対応します。これについては次のセクションで話しましょう。

24.7 チャレンジ

1. 一変数の多項式の環の評価関数を実装してください。例えば、
 $4x^2 - 1$ は (ゼロ次のべきで始まる) `[-1, 0, 4]` として表される
ような、 x のべきの前の係数のリストとして多項式を表現でき
ます。
2. 前の構造を多くの独立変数の多項式に一般化してください、例
えば $x^2y - 3y^3z$ 。
3. 2×2 行列の環の代数を実装してください。
4. 自然数の平方のリストを生成する余代数の定義。
5. `unfoldr` を使用して最初の n 個の素数のリストを生成してくだ
さい。

25

モナドに関する代数

もし自己関手を式の定義方法として解釈するならば、代数はそれらを評価する方法を提供し、モナドはそれらを形成し操作する方法を提供します。代数とモナドを組み合わせることで、多くの機能を得られるだけでなく、いくつかの興味深い問い合わせにも答えることができます。

そのような問い合わせの一つは、モナドと隨伴の関係についてです。見てきたように、すべての隨伴はモナド（および余モナド）を定義します。問題は、すべてのモナド（余モナド）が隨伴から導出されるかどうかです。この答えは肯定的です。与えられたモナドを生成する隨伴の全ての族が存在します。私はそのような隨伴を二つ示します。



まず定義を見直しましょう。モナドは自己関手 m で、いくつかの自然変換とそれらが満たすべき整合性条件で装備されています。これらの変換のコンポーネントは、 a において以下のようにになります：

$$\begin{aligned}\eta_a &:: a \rightarrow m\ a \\ \mu_a &:: m(m\ a) \rightarrow m\ a\end{aligned}$$

同じ自己関手のための代数は、特定の対象—担い手 $a-$ と射：

$$alg :: m\ a \rightarrow a$$

を選択するものです。最初に気づくべきは、代数が η_a とは逆の方向に進むということです。直感的には、 η_a は型 a の値からトリビアルな式を作ります。代数をモナドと互換性のあるようにする最初の整合性条件は、担い手が a である代数を使ってこの式を評価すると、元の値が得られるということです：

$$alg \circ \eta_a = \mathbf{id}_a$$

二番目の条件は、二重にネストされた式 $m(m\ a)$ を評価する二つの方法があるという事実から生じます。まず μ_a を適用して式を平らにし、

その後代数の評価子を使うか、あるいは持ち上げられた評価子を適用して内部の式を評価し、その結果に評価子を適用するかです。二つの戦略が等価であることが望ましいです:

$$alg \circ \mu_a = alg \circ m\ alg$$

ここで、`m alg` は関手 `m` を使って `alg` から持ち上げられた射です。次の交換図は二つの条件を記述します (以下では何が来るかを予測して `m` を `T` に置き換えていません) :

$$\begin{array}{ccc} a & \xrightarrow{\eta_a} & Ta \\ & \searrow \text{alg} & \downarrow \\ & a & \end{array} \quad \begin{array}{ccc} T(Ta) & \xrightarrow{T alg} & Ta \\ \downarrow \mu_a & & \downarrow \\ Ta & \xrightarrow{alg} & a \end{array}$$

これらの条件を Haskell で表現することもできます。

```
alg . return = id
alg . join = alg . fmap alg

alg compose pure == id
alg compose flatten == alg compose fmap(alg)
```

小さな例を見てみましょう。リスト自己関手の代数は、型 `a` と、`a` のリストから `a` を生成する関数で構成されます。この関数は `foldr` を使って表現することができます。ここで要素の型も蓄積器の型も `a` に等しいと選択します:

```
foldr :: (a -> a -> a) -> a -> [a] -> a
```

```
def foldr[A]: (A => A => A) => A => List[A] => A
```

この特定の代数は二引数の関数、それを `f` と呼びましょう、と値 `z` によって指定されます。リスト関手は偶然にもモナドで、`return` は値を单一のリストに変換します。代数の合成、ここでは `foldr f z`、`return` の後に適用されると、`x` を以下に変換します：

```
foldr f z [x] = x `f` z
```

```
def foldr[A]: (A => A => A) => A => List[A] => A =
  f => z => {
    case x :: Nil => f(x)(z)
  }
```

ここで `f` の作用は中置記法で書かれています。代数がモナドと互換性がある場合、次の整合性条件がすべての `x` に対して満たされます：

```
x `f` z = x
```

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

`f` を二項演算子と見るならば、この条件は `z` が右単位要素であることを教えてくれます。

二番目の整合性条件はリストのリストに作用します。`join` の作用は個々のリストを連結します。結果のリストを畳み込むことができます。あるいは、個々のリストを最初に畳み込み、その結果のリストを畳み込むこともできます。再び、`f` を二項演算子と解釈すると、この条件はこの二項演算が結合的であることを教えてくれます。これらの条件は、 (a, f, z) がモノイドであるとき確かに満たされます。

25.1 T-代数

数学者が自分たちのモナドを T と呼ぶのを好むので、それらと互換性のある代数を T-代数と呼びます。ある圏 C の中で与えられたモナド T の T-代数は、Eilenberg-Moore 圏と呼ばれることが多い圏を形成し、通常 C^T と表記されます。その圏の射は代数の準同型です。これらは F-代数に定義されたのと同じ準同型です。

T-代数は担い手対象と評価子のペアです、 (a, f) 。明らかに C^T から C への忘却関手 U^T があります。これは (a, f) を a に写します。また T-代数の準同型を C の対応する担い手対象間の射に写します。随伴についての議論から覚えているかもしれません、忘却関手の左随伴は自由関手と呼ばれます。

U^T の左随伴は F^T と呼ばれます。これは C の対象 a を C^T の自由代数に写します。この自由代数の担い手は Ta です。その評価子は $T(Ta)$ から Ta への射です。 T がモナドであるので、モナドの μ_a (Haskell では `join`) を評価子として使用できます。

これが T-代数であることを示すためには、二つの整合性条件が満たされている必要があります:

$$\begin{aligned} \textit{alg} \circ \eta_{Ta} &= \mathbf{id}_{Ta} \\ \textit{alg} \circ \mu_a &= \textit{alg} \circ T \textit{ alg} \end{aligned}$$

しかし、これらは代数に μ を代入した場合のモナド則にすぎません。

すでに見たように、すべての随伴はモナドを定義します。実は F^T と U^T の間の随伴は、その構成に使用されたモナド T そのものを定義します。すべてのモナドに対してこの構成を行うことができるので、すべてのモナドは随伴から生成されると結論づけることができます。後で、同じモナドを生成する別の随伴を示します。

計画は次の通りです。まず F^T が実際に U^T の左随伴であることを示します。これを行うには、この随伴の単位と余単位を定義し、対応する三角恒等式が満たされていることを証明します。次に、この随伴によって生成されたモナドが実際に元のモナドであることを示します。

随伴の単位は自然変換です:

$$\eta :: I \rightarrow U^T \circ F^T$$

この変換の a コンポーネントを計算しましょう。恒等関手は私たちに a を与えます。自由関手は自由代数 (Ta, μ_a) を生成し、忘却関手はそれを Ta に減らします。合わせると、 a から Ta への写像を得ます。単純にモナド T の単位をこの随伴の単位として使用します。

余単位を見てみましょう:

$$\varepsilon :: F^T \circ U^T \rightarrow I$$

この T-代数 (a, f) でのコンポーネントを計算しましょう。忘却関手は f を忘れ、自由関手はペア (Ta, μ_a) を生成します。したがって、余単位 ε のコンポーネントを T-代数 (a, f) で定義するためには、Eilenberg-Moore 圏内の正しい射、または T-代数の準同型が必要です:

$$(Ta, \mu_a) \rightarrow (a, f)$$

このような準同型は担い手 Ta を a に写すべきです。忘れられた評価子 f を復活させましょう。今度は T-代数の準同型としてそれを使用します。実際に、 f が T-代数であることを示すのと同じ交換図は、それが T-代数の準同型であることを再解釈することで示すことができます:

$$\begin{array}{ccc} T(Ta) & \xrightarrow{Tf} & Ta \\ \downarrow \mu_a & & \downarrow f \\ Ta & \xrightarrow{f} & a \end{array}$$

これにより、T-代数圏の対象 (a, f) での余単位自然変換 ε のコンポーネントを f として定義しました。

単位と余単位が三角恒等式を満たしていることを示すためには、それらも証明する必要があります。これらは次のようになります:

$$\begin{array}{ccc} Ta & \xrightarrow{T\eta_a} & T(Ta) \\ & \searrow & \downarrow \mu_a \\ & & Ta \end{array} \quad \begin{array}{ccc} a & \xrightarrow{\eta_a} & Ta \\ & \searrow & \downarrow f \\ & & a \end{array}$$

最初の恒等式はモナド T の単位則によって支持されます。二番目のは T-代数 (a, f) の規則です。

これにより、二つの関手が随伴を形成することが確立されました:

$$F^T \dashv U^T$$

すべての随伴はモナドを引き起こします。往復

$$U^T \circ F^T$$

は、対応するモナドを生じさせる \mathbf{C} の自己関手です。対象 a に対するその作用を見てみましょう。 F^T によって作成された自由代数は (Ta, μ_a) です。忘却関手 U^T は評価子を落とします。したがって、確かに:

$$U^T \circ F^T = T$$

予想通り、随伴の単位はモナド T の単位です。

随伴の余単位がモナドの乗算を次の式を通じて生成することを覚えておくかもしれません:

$$\mu = R \circ \varepsilon \circ L$$

これは、それぞれ L から L へ、 R から R への恒等自然変換である二つの自然変換と、中央にある余単位である三つの自然変換の水平合成です。余単位は、T-代数 (a, f) でのコンポーネントが f である自然変換です。

μ_a のコンポーネントを計算しましょう。最初に F^T の後に ε を水平合成します。これは $F^T a$ での ε のコンポーネントを結果とします。 F^T

が a を代数 (Ta, μ_a) に写し、 ε が評価子を選ぶので、 μ_a で終わります。左側に U^T との水平合成は何も変えません、なぜなら U^T の射の作用は自明だからです。したがって、確かに、随伴から得られる μ は元のモナド T の μ と同じです。

25.2 Kleisli 圈

以前に Kleisli 圈について見ました。これは他の圏 C とモナド T から構成される圏です。この圏を C_T と呼びます。Kleisli 圈 C_T の対象は C の対象ですが、射は異なります。Kleisli 圈の射 f_K は、オリジナルの圏で a から Tb への射 f に対応します。この射を a から b への Kleisli 射と呼びます。

Kleisli 圈の射の合成は、Kleisli 射のモナド的合成に関して定義されます。例えば、 g_K を f_K の後に合成しましょう。Kleisli 圈では:

$$\begin{aligned} f_K &:: a \rightarrow b \\ g_K &:: b \rightarrow c \end{aligned}$$

これは、圏 C では以下に対応します:

$$\begin{aligned} f &:: a \rightarrow Tb \\ g &:: b \rightarrow Tc \end{aligned}$$

合成を定義します:

$$h_K = g_K \circ f_K$$

C の Kleisli 射として

$$\begin{aligned} h &:: a \rightarrow Tc \\ h &= \mu \circ (Tg) \circ f \end{aligned}$$

Haskell では次のように書きます:

```
h = join . fmap g . f  
h == flatten.compose(fmap(g) _ compose f)
```

C から C_T への関手 F があり、これは対象に対して自明に作用します。射に対しては、 C の射 f を、 f の戻り値を飾る Kleisli 射を作成することによって C_T の射に写します。射が与えられたとき:

$$f :: a \rightarrow b$$

それは、対応する Kleisli 射を持つ C_T の射を作成します:

$$\eta \circ f$$

Haskell では次のように書きます:

```
return . f  
unit compose f
```

また \mathbf{C}_T から \mathbf{C} への関手 G を定義することもできます。これは Kleisli 圏の対象 a を \mathbf{C} の対象 Ta に写します。Kleisli 射に対応する射 $f_{\mathbf{K}}$:

$$f :: a \rightarrow Tb$$

は、 \mathbf{C} の射として

$$Ta \rightarrow Tb$$

です。これは最初に f を持ち上げてから μ を適用することによって与えられます:

$$\mu_{Tb} \circ Tf$$

Haskell の表記ではこれは次のようにになります:

```
G f_T = join . fmap f
```

これは、`join`を使ってモナド的な束縛を定義する方法として認識できます。

二つの関手が随伴を形成することは容易に見て取れます:

$$F \dashv G$$

そして、それらの合成 $G \circ F$ は元のモナド T を再生産します。

これが同じモナドを生成する二番目の随伴です。実際、同じモナド T を結果とする随伴の全体の圏 $\mathbf{Adj}(\mathbf{C}, T)$ があります。先ほど見た Kleisli 随伴はこの圏の始対象であり、Eilenberg-Moore 随伴は終対象です。

25.3 余代数と余モナド

余モナド W に対しても類似の構成が行われます。私たちは余代数の圏を定義することができ、それは余モナドと互換性のある余代数です。それらは次の図を可換にします:

$$\begin{array}{ccc} a & \xleftarrow{\epsilon_a} & Wa \\ & \nearrow coa & \uparrow a \\ & & Wa \end{array} \quad \begin{array}{ccc} W(Wa) & \xleftarrow{W coa} & Wa \\ \delta_a \uparrow & & coa \uparrow \\ Wa & \xleftarrow{coa} & a \end{array}$$

ここで coa は余代数の余評価射で、その担い手は a です:

$$coa :: a \rightarrow Wa$$

そして ϵ と δ は余モナドを定義する二つの自然変換です (Haskell では、それらのコンポーネントは **extract** と **duplicate** と呼ばれます)。

これらの余代数から C への明らかな忘却関手 U^W があります。これは単に余評価を忘れます。右随伴 F^W を考えます。

$$U^W \dashv F^W$$

忘却関手の右随伴は余自由関手と呼ばれます。 F^W は C の対象 a に対して、余代数 (Wa, δ_a) を割り当てます。随伴は合成 $U^W \circ F^W$ として元の余モナドを再生産します。

同様に、余 Kleisli 圏を構成し、対応する随伴から余モナドを再生成することができます。

25.4 レンズ

レンズに関する議論に戻りましょう。レンズは余代数として書くことができます:

$$\text{coalg}_s :: a \rightarrow \text{Store } s a$$

関手 $\text{Store } s$ に対して:

```
data Store s a = Store (s -> a) s

case class Store[S, A](run: S => A, s: S)

// convenient partially applied constructor
object Store {
  def apply[S, A](run: S => A): S => Store[S, A] =
    s => new Store(run, s)
}
```

この余代数は、関数のペアとしても表現できます:

$$\begin{aligned} set &:: a \rightarrow s \rightarrow a \\ get &:: a \rightarrow s \end{aligned}$$

(ここで a は「全部」と考え、 s はその「小さい」部分としてください。) このペアの観点から、私たちは持っています:

$$\text{coalg}_s a = \text{Store} (\text{set } a) (\text{get } a)$$

ここで、 a は型 a の値です。部分適用された **set** は関数 $s \rightarrow a$ です。

また、*Store s* が余モナドであることも知っています:

```
instance Comonad (Store s) where
    extract (Store f s) = f s
    duplicate (Store f s) = Store (Store f) s

implicit def storeComonad[S] = new Comonad[Store[S, ?]] {
    def extract[A](wa: Store[S, A]): A = wa match {
        case Store(f, s) => f(s)
    }

    def duplicate[A](wa: Store[S, A]): Store[S, Store[S, A]] = wa match {
        case Store(f, s) => Store(Store(f), s)
    }
}
```

問題は、レンズがこの余モナドのための余代数である条件は何かということです。最初の整合性条件:

$$\varepsilon_a \circ coalg = \mathbf{id}_a$$

は次のように翻訳されます:

$$set\ a\ (get\ a) = a$$

これはレンズの規則で、構造体 a のフィールドを以前の値に設定しても何も変わらないという事実を表しています。

二番目の条件:

$$fmap\ coalg \circ\ coalg = \delta_a \circ\ coalg$$

はもう少し作業が必要です。まず、**Store** 関手の **fmap** の定義を思い出してください:

```
fmap g (Store f s) = Store (g . f) s

implicit def storeFunctor[S] = new Functor[Store[S, ?]] {
  def fmap[A, B](g: A => B)(fa: Store[S, A]): Store[S, B] = fa match {
    case Store(f, s) =>
      Store(g compose f, s)
  }
}
```

fmap coalg を **coalg** の結果に適用すると、私たちは持っています:

```
Store (coalg . set a) (get a)

Store(coalg compose set(a), get(a))
```

一方、**duplicate** を **coalg** の結果に適用すると、生じるのは:

```
Store (Store (set a)) (get a)
```

```
Store(Store(set(a)), get(a))
```

これら二つの式が等しいためには、`Store` の下の二つの関数が任意の `s` に作用するとき等しくなければなりません:

```
coalg (set a s) = Store (set a) s
```

```
coalg(set(a)(s)) == Store(set(a))(s)
```

`coalg` を展開すると、私たちは持っています:

```
Store (set (set a s)) (get (set a s)) = Store (set a) s
```

```
Store(set(set(a)(s)))(get(set(a)(s))) == Store(set(a))(s)
```

これは残りの二つのレンズ則に相当します。最初の規則:

```
set (set a s) = set a
```

```
set(set(a)(s)) == set(a)
```

は、フィールドの値を二回設定するのと一回設定するのが同じであると言っています。二番目の規則:

```
get (set a s) = s
```

```
get(set(a)(s)) == s
```

は、 s に設定されたフィールドの値を取得すると、 s が戻ってくると言っています。

言い換えれば、行儀の良いレンズは確かに **Store** 関手のための余モナド余代数です。

25.5 チャレンジ

1. 自由関手 $F :: C \rightarrow C^T$ の射に対する作用は何か。ヒント: モナドの μ の自然性条件を使用します。
2. 随伴を定義してください:

$$U^W \dashv F^W$$

3. 上記の随伴が元の余モナドを再現することを証明してください。

26

エンドと余エンド

巻における射に対して様々な直観を持つことができますが、対象 a から対象 b への射がある場合、両対象は何らかの方法で「関連している」ということについては全員が同意するでしょう。射は、この関係の証明であるという意味で、任意の半順序集合の圏では、射が関係そのものです。一般的に、二つの対象間の同じ関係の「証明」は多数存在することがあります。これらの証明は、ホム集合と呼ばれる集合を形成します。対象が変わると、対象のペアから「証明」の集合へのマッピングが得られます。このマッピングは関手的であり、第一引数に反変で第二引数に共変です。これを圏の対象間の大局的な関係を確立していると見なすことができます。この関係はホム関手によっ

て記述されます:

$$C(-,=) :: C^{op} \times C \rightarrow \text{Set}$$

一般に、このような関手は、圏の対象間の関係を確立すると解釈することができます。関係は、異なる二つの圏 C と D を巻き込むこともあります。このような関係を記述する関手は、次のシグネチャを持ち、プロ関手と呼ばれます:

$$p :: D^{op} \times C \rightarrow \text{Set}$$

数学者は、これを C から D へのプロ関手(注目してください、反転しています)と言います、そのシンボルとして斜線が付いた矢印を使います:

$$C \not\rightarrow D$$

プロ関手は、 C の対象と D の対象の間の証明関連関係と考えることができます。ここで、集合の要素は関係の証明を象徴しています。いつ $p a b$ が空であるならば、 a と b の間には関係がありません。関係は対称的である必要はないことに注意してください。

もう一つの有用な直観は、自己関手がコンテナであるという考え方の一般化です。プロ関手の値 $p a b$ は、型 a の要素によってキー付けされた b のコンテナと見なすことができます。特に、ホムプロ関手の要素は a から b への関数です。

Haskell では、プロ関手は二引数の型コンストラクタ \mathbf{p} と、ペアの関数を持ち上げると呼ばれるメソッド \mathbf{dimap} で定義されます。最初の関数は「間違った方向」に行きます:

```
class Profunctor p where
    dimap :: (c -> a) -> (b -> d) -> p a b -> p c d

trait Profunctor[P[_], _] {
    def dimap[A, B, C, D]
        (f: C => A)(g: B => D)(pab: P[A, B]): P[C, D]
}
```

プロ関手の関手性は、もし a が b に関連している証明があれば、 c が d に関連している証明を得ることができます。ただし、 c から a への射と、 b から d への別の射がある場合です。あるいは、最初の関数を新しいキーを古いキーに翻訳するものとし、二つ目の関数をコンテナの内容を変更するものとして考えることもできます。

一つの圏内で作用するプロ関手については、型 $p a a$ の対角要素からかなりの情報を抽出することができます。射のペア $b \rightarrow a$ と $a \rightarrow c$ がある限り、 b が c に関連していることを証明することができます。さらに良いことに、単一の射を使って対角線の値に達することができます。例えば、射 $f :: a \rightarrow b$ がある場合、ペア $\langle f, \text{id}_b \rangle$ を持ち上げて $p b b$ から $p a b$ へ行くことができます:

```
dimap f id (p b b) :: p a b

dimap(f)(identity[B])(pbb): P[A, B]
```

または、ペア $\langle \text{id}_a, f \rangle$ を持ち上げて $p\ a\ a$ から $p\ a\ b$ へ行くことができます:

```
dimap id f (p a a) :: p a b
```

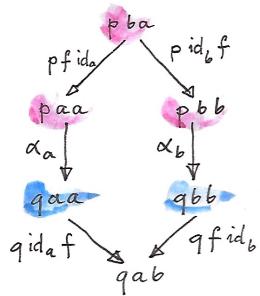
```
dimap(identity[A])(f)(paa): P[A, B]
```

26.1 双自然変換

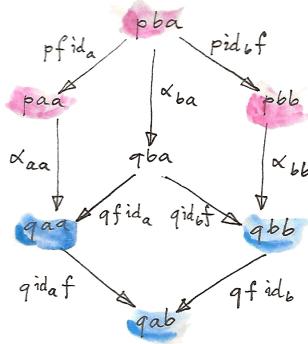
プロ関手は関手なので、標準的な方法でそれらの間の自然変換を定義することができます。多くの場合、二つのプロ関手の対角要素の間のマッピングを定義するだけで十分です。このような変換は、対角要素を非対角要素に接続する二つの方法を反映した可換性条件を満たす場合、双自然変換と呼ばれます。プロ関手 p と q の間の双自然変換は、関手圏 $[C^{op} \times C, \text{Set}]$ のメンバーである変換の形態族です:

$$\alpha_a :: p\ a\ a \rightarrow q\ a\ a$$

以下の図の通り、任意の射 $f :: a \rightarrow b$ に対して以下の図が可換であること:



この条件は自然性条件よりも明らかに弱いです。もし α が $[C^{op} \times C, \text{Set}]$ における自然変換であれば、上記の図は二つの自然性の四角形と一つの関手性条件（プロ関手 q が合成を保存する）から構成されます：



$[C^{op} \times C, \text{Set}]$ における自然変換 α のコンポーネントは対象のペアによってインデックス付けされます α_{ab} 。一方、双自然変換は、それが対

応するプロ関手の対角要素のみをマッピングするので、一つの対象によってインデックス付けされます。

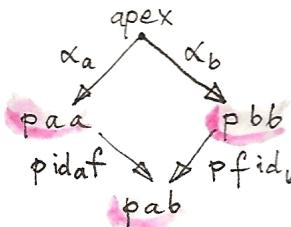
26.2 エンド

「代数」から「微積分」に進む準備ができました。エンド (および余エンド) の微積分は、伝統的な微積分からアイデアといいくつかの表記を借りています。特に、余エンドは無限和または積分として理解されるかもしれません、エンドは無限積に似ています。Dirac のデルタ関数のようなものさえあります。

エンドは極限の一般化であり、関手に代わってプロ関手があります。錐体に代わって、私たちはくさびを持っています。くさびの底はプロ関手 p の対角要素によって形成されます。くさびの頂点は対象 (ここでは、**Set** 値のプロ関手を考慮しているので、集合) であり、側面は頂点から底部の集合への関数の族です。この族を一つの多相的関数と考えることができます — その戻り型が多相的な関数です:

$$\alpha :: \forall a . \text{apex} \rightarrow p a a$$

錐体とは異なり、くさび内には基部の頂点を接続する関数はありません。しかし、私たちが以前見たように、C 中の任意の射 $f :: a \rightarrow b$ を与えられると、私たちは $p a a$ と $p b b$ の両方を共通の集合 $p a b$ に接続することができます。したがって、次の図が可換であることを主張します:



これはくさび条件と呼ばれます。これは次のように書くことができます:

$$p \text{id}_a f \circ \alpha_a = p f \text{id}_b \circ \alpha_b$$

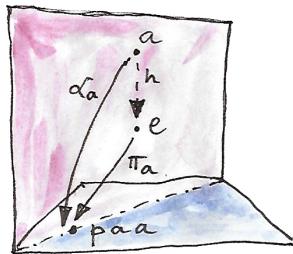
あるいは、Haskell 表記を使って:

```
dimap id f . alpha = dimap f id . alpha

(dimap(identity[A])(f) _ compose alpha) ==
(dimap(f)(identity[B] _ compose alpha))
```

これで私たちは、普遍構成を用いて、 p のエンドを普遍くさびとして定義することができます — 一つの集合 e と一つの関数の族 π を持つもので、任意の他のくさびが頂点 a と一つの族 α を持つ場合、全ての三角形を可換にする一意な関数 $h :: a \rightarrow e$ が存在します:

$$\pi_a \circ h = \alpha_a$$



エンドのシンボルは積分符号であり、「積分変数」は添字位置にあります:

$$\int_c pcc$$

π のコンポーネントはエンドのための射影マップと呼ばれます:

$$\pi_a :: \int_c pcc \rightarrow paa$$

C が離散圏 (恒等射以外の射がない) である場合、エンドは p の全ての対角要素の全体的な積です。後で、もう少し一般的なケースでは、エンドとこの積との間にイコライザを通じて関係があることを示します。

Haskell では、エンドの式は全称量化子に直接翻訳されます:

```

forall a. p a a

// no Rank-N types in Scala
// have to introduce a polymorphic function
trait PolyFunction1[P[_, _]] {
  def apply[A](): P[A, A]
}

```

```
}
```

厳密に言えば、これは p の全ての対角要素のただの積ですが、くさび条件はパラメトリシティ^{*1}により自動的に満たされます。任意の関数 $f :: a \rightarrow b$ に対して、くさび条件は以下のようになります:

```
dimap f id . pi = dimap id f . pi
```

```
(dimap(f)(identity[B]) _ compose pi.apply) ==
(dimap(identity[A])(f) _ compose pi.apply)
```

または、型注釈付きで:

```
dimap f idb . pib = dimap ida f . pia
```

ここで、方程式の両側は型:

```
Profunctor p => (forall c. p c c) -> p a b
```

```
def side[P[_ , _]: Profunctor]: PolyFunction1[P] => P[A, B]
```

を持ち、`pi` は多相的射影です:

```
pi :: Profunctor p => forall c. (forall a. p a a) -> p c c
pi e = e
```

^{*1} <https://bartoszmilewski.com/2017/04/11/profunctor-parametricity/>

```
// no Rank-N types in Scala
// need a higher rank polymorphic function
trait PolyFunction2[P[_, _]] {
  def apply[C](in: PolyFunction1[P]): P[C, C]
}

def pi[P[_, _]](implicit P: Profunctor[P]): PolyFunction2[P] =
  new PolyFunction2[P] {
    def apply[C](in: PolyFunction1[P]): P[C, C] =
      in()
}
```

ここで、型推論は自動的に `e` の正しいコンポーネントを選択します。

錐体のための全ての可換性条件を一つの自然変換として表現することができたのと同様に、くさび条件を一つの双自然変換にまとめるすることができます。そのためには、全ての対象のペアを单一の対象 c にマッピングし、全ての射のペアをこの対象の恒等射にマッピングする定数関手 Δ_c への定数プロ関手の一般化が必要です。くさびはその関手からプロ関手 p への双自然変換です。実際には、双自然変換の六角形は、 Δ_c が全ての射を一つの恒等関数に持ち上げるときにくさびダイアモンドに縮小します。

エンドは `Set` 以外のターゲット圏に対しても定義することができますが、ここでは `Set`-値のプロ関手とそのエンドのみを考慮します。

26.3 エンドとイコライザとして

エンドの定義における可換性条件はイコライザを使って書くことができます。まず、Haskell の表記を使用して(数学的な表記はこの場合ユーザーフレンドリーでないようです)、くさび条件の二つの収束する枝に対応する二つの関数を定義しましょう:

```
lambda :: Profunctor p => p a a -> (a -> b) -> p a b  
lambda paa f = dimap id f paa
```

```
rho :: Profunctor p => p b b -> (a -> b) -> p a b  
rho pbb f = dimap f id pbb
```

```
def lambda[A, B, P[_]](P: Profunctor[P]): P[A, A] => (A =>  
  B) => P[A, B] =  
  paa => f => P.dimap(identity[A])(f)(paa)
```

```
def rho[A, B, P[_]](P: Profunctor[P]): P[B, B] => (A => B)  
  => P[A, B] =  
  pbb => f => P.dimap(f)(identity[B])(pbb)
```

両方の関数はプロ関手 p の対角要素を型:

```
type ProdP p = forall a b. (a -> b) -> p a b
```

```
trait ProdP[P[_, _]] {
    def apply[A, B](f: A => B): P[A, B]
}
```

の多相的関数にマップします。

これらの関数は異なる型を持っています。しかし、`p` の全ての対角要素を集めることで一つの大規模な積型を形成することにより、その型を統一することができます:

```
newtype DiaProd p = DiaProd (forall a. p a a)

case class DiaProd[P[_, _]](paa: PolyFunction1[P])
```

関数 `lambda` と `rho` はこの積型から二つのマッピングを誘導します:

```
lambdaP :: Profunctor p => DiaProd p -> ProdP p
lambdaP (DiaProd paa) = lambda paa
```

```
rhoP :: Profunctor p => DiaProd p -> ProdP p
rhoP (DiaProd pbb) = rho pbb
```

```
def lambdaP[P[_, _]](P: Profunctor[P]): DiaProd[P] => ProdP[P] = {
    case DiaProd(paa) =>
        new ProdP[P] {
            def apply[A, B](f: A => B): P[A, B] =
                lambda(P)(paa[A])(f)
        }
}
```

```
        }
    }

def rhoP[P[_, _]](P: Profunctor[P]): DiaProd[P] => ProdP[P] = {
    case DiaProd(paa) =>
        new ProdP[P] {
            def apply[A, B](f: A => B): P[A, B] =
                rho(P)(paa[B])(f)
        }
}
```

`p` のエンドはこれら二つの関数のイコライザです。イコライザは二つの関数が等しい最大の部分集合を選び出します。この場合、それはくさびダイアグラムが可換になるための `p` の全ての対角要素の積の部分集合を選び出します。

26.4 自然変換としてのエンド

エンドの最も重要な例は、自然変換の集合です。二つの関手 F と G の間の自然変換は、形式 $C(Fa, Ga)$ のホム集合から選ばれる射の族です。自然性条件がなければ、自然変換の集合はこれらのホム集合の全ての積に過ぎません。実際、Haskell ではそれがそうです:

```
forall a. f a -> g a
```

```

// Yet another type needs to be introduced.
// To read more about FunctionK (~>):
// typelevel.org/cats/datatypes/functionk.html
trait ~>[F[_], G[_]] {
  def apply[B](fa: F[B]): G[B]
}

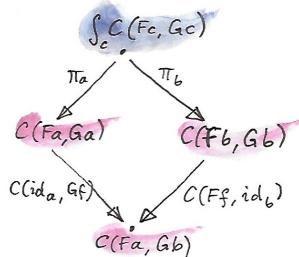
```

$F \rightsquigarrow G$

Haskell でうまくいく理由は、自然性がパラメトリシティによって従うからです。しかし、Haskell の外では、そのようなホム集合を横断する全ての対角セクションが自然変換を生み出すわけではありません。しかし、注意してください:

$$\langle a, b \rangle \rightarrow C(Fa, Gb)$$

はプロ関手なので、そのエンドを研究することは意味があります。これがくさび条件です:



集合 $\int_c \mathbf{C}(Fc, Gc)$ から一つの要素を選び出しましょう。二つの射影はこの要素を特定の変換の二つのコンポーネントにマップします。それらを呼びましょう:

$$\begin{aligned}\tau_a &:: Fa \rightarrow Ga \\ \tau_b &:: Fb \rightarrow Gb\end{aligned}$$

左の枝では、ペアの射 $\langle \mathbf{id}_a, Gf \rangle$ をホム関手で持ち上げます。そのような持ち上げは同時に前置きと後置きの合成として実装されることを思い出してください。 τ_a に作用する持ち上げられたペアは次のようになります:

$$Gf \circ \tau_a \circ \mathbf{id}_a$$

図の他の枝は次のようになります:

$$\mathbf{id}_b \circ \tau_b \circ Ff$$

くさび条件によって要求される彼らの等式は、 τ の自然性条件に他なりません。

26.5 余エンド

予想通り、エンドの双対は余エンドと呼ばれます。これは余くさび(cowedge、コウェッジと発音されます、カウエッジではありません)と呼ばれるくさびの双対から構成されます。



エッジの効いた牛？

余エンドのシンボルは積分符号で、「積分変数」は上付き文字の位置にあります:

$$\int^c pcc$$

エンドが積に関連しているように、余エンドは余積、つまり和(この点で、それは積分に似ています、それは和の極限です)に関連しています。射影の代わりに、私たちはプロ関手の対角要素から余エンドへの入射を持っています。もしくさび条件がなければ、プロ関手 p の余エンドは paa 、または pbb 、または pcc などのいずれかであり、またはそのような a が存在し、余エンドは単に集合 paa であると言ることができます。エンドの定義に使用された全称量化子は余エンドに対する存在量化子になります。

このため、擬似 Haskell では、余エンドを次のように定義します:

```
exists a. p a a
```

存在量化子を Haskell でエンコードする標準的な方法は、全称量化されたデータコンストラクタを使用することです。したがって、定義することができます:

```
data Coend p = forall a. Coend (p a a)

trait Coend[P[_], _] {
  def paa[A]: P[A, A]
}
```

この背後にある論理は、任意の族の型 $p a a$ の値を使用して余エンドを構成できるはずです、私たちが選んだどんな a に関係なく。

エンドがイコライザを使って定義できるように、余エンドも余イコライザを使って記述できます。全てのくさび条件は、 $p a b$ の全ての可能な関数 $b \rightarrow a$ に対する一つの巨大な余積を取ることで要約できます。Haskell では、それは実存型として表現されます:

```
data SumP p = forall a b. SumP (b -> a) (p a b)

trait SumP[P[_], _] {
  def f[A, B]: B => A
  def pab[A, B]: P[A, B]
}
```

この和型は、`dimap` を使用して関数を持ち上げ、それをプロ関手 p に適用することで二つの方法で評価できます:

```

lambda, rho :: Profunctor p => SumP p -> DiagSum p
lambda (SumP f pab) = DiagSum (dimap f id pab)
rho     (SumP f pab) = DiagSum (dimap id f pab)

def lambda[P[- _, _]](P: Profunctor[P]): SumP[P] => DiagSum[P] =
  sump => new DiagSum[P] {
    def paa[A]: P[A, A] =
      P.dimap(sump.f)(identity[A])(sump.pab)
  }

def rho[P[_, _]](P: Profunctor[P]): SumP[P] => DiagSum[P] =
  sump => new DiagSum[P] {
    def paa[A]: P[A, A] =
      P.dimap(identity[A])(sump.f)(sump.pab)
  }

```

ここで `DiagSum` は p の対角要素の和です:

```

data DiagSum p = forall a. DiagSum (p a a)

trait DiagSum[P[_, _]]{
  def paa[A]: P[A, A]
}

```

これら二つの関数の余イコライザは余エンドです。余イコライザは `DiagSum p` から、`lambda` または `rho` を同じ引数に適用することで得られる値を識別することによって得られます。引数は関数 $b \rightarrow a$ と要

素 pab のペアです。`lambda` と `rho` の適用は、型 `DiagSum p` の二つの潜在的に異なる値を生成します。余エンドでは、これら二つの値は識別され、くさび条件が自動的に満たされます。

関連する要素を集合で識別するプロセスは、形式的には商を取ることとして知られています。商を定義するためには、**同値関係** \sim が必要です。関係は反射的、対称的、推移的でなければなりません：

$$a \sim a$$

$$\text{もし } a \sim b \text{ ならば } b \sim a$$

$$\text{もし } a \sim b \text{ かつ } b \sim c \text{ ならば } a \sim c$$

このような関係は、集合を同値クラスに分割します。各クラスは互いに関連する要素で構成されます。我々は、各クラスから代表を選んで商集合を形成します。典型的な例は、以下の同値関係を持つ整数のペアの定義による有理数の定義です：

$$(a, b) \sim (c, d) \text{ iff } a * d = b * c$$

これが同値関係であることは簡単に確認できます。ペア (a, b) は分数 $\frac{a}{b}$ として解釈され、分子と分母に共通の除数がある分数は識別されます。有理数はそのような分数の同値クラスです。

我々は以前の極限と余極限の議論から、ホム関手が連続である、つまり極限を保存することを思い出してください。双対的に、反変ホム関手は余極限を極限に変換します。これらの性質はエンドと余エンドに一般化されます。エンドと余エンドはそれぞれ極限と余極限の一般

化です。特に、次のような非常に便利な恒等式が得られます:

$$\text{Set}\left(\int^x p\ x\ x, c\right) \cong \int_x \text{Set}(p\ x\ x, c)$$

これを擬似 Haskell で見てみましょう:

```
(exists x. p x x) -> c ≡ forall x. p x x -> c
```

これは、実存型を取る関数が多相的関数と等価であることを教えてくれます。そのような関数は、実存型にエンコードされている可能性のある任意の型を処理する準備ができていなければなりません。これは、和型を受け入れる関数がケース文として実装されなければならず、和に存在する各型に対するハンドラのタプルを持つという同じ原理です。ここでは、和型が余エンドに置き換えられ、ハンドラの族はエンド、または多相的関数になります。

26.6 忍者米田の補題

米田の補題に出てくる自然変換の集合は、エンドを使用して次のようにエンコードできます:

$$\int_z \text{Set}(\mathbf{C}(a, z), Fz) \cong Fa$$

また、双対の式もあります:

$$\int^z \mathbf{C}(z, a) \times Fz \cong Fa$$

この恒等式は、Dirac のデルタ関数 ($a = z$ で無限大のピークを持つ関数 $\delta(a - z)$ 、またはむしろ超関数) の式を強く思い起こさせます。ここでは、ホム関手がデルタ関数の役割を果たします。

これら二つの恒等式は、時々忍者米田の補題と呼ばれます。

第二の式を証明するために、任意の対象 c へ行くホム関手の中に証明したい恒等式の左側を挿入します:

$$\mathbf{Set}\left(\int^z \mathbf{C}(z, a) \times Fz, c\right)$$

連続性の議論を使用して、余エンドをエンドに置き換えることができます:

$$\int_z \mathbf{Set}(\mathbf{C}(z, a) \times Fz, c)$$

これで、積と指數関数の間の随伴を利用することができます:

$$\int_z \mathbf{Set}(\mathbf{C}(z, a), c^{(Fz)})$$

今、積分を「実行」することができます。米田の補題を使用して次のように得られます:

$$c^{(Fa)}$$

(ここでは、関手 $c^{(Fz)}$ が z において反変であるため、米田の補題の反変版を使用しました。) この指數関数対象はホム集合に同型です:

$$\mathbf{Set}(Fa, c)$$

最後に、米田の埋め込みを利用して、次の同型に到達します:

$$\int^z \mathbf{C}(z, a) \times Fz \cong Fa$$

26.7 プロ関手の合成

プロ関手が関係を記述する、より正確には証明関連関係を記述するというアイデアをさらに探求しましょう。集合 pab は a が b に関連している証明の集合を表します。もし私たちが二つの関係 p と q を持っているなら、それらを合成してみることができます。 a が q の後に p を介して b に関連していると言います。もし中間対象 c が存在し、両方の qbc と PCA が空でない場合です。この新しい関係の証明は個々の関係の証明のペアです。したがって、存在量化子が余エンドに対応し、二つの集合のデカルト積が「証明のペア」に対応することを理解すると、次の式を使用してプロ関手の合成を定義できます:

$$(q \circ p) ab = \int^c PCA \times qbc$$

これが `Data.Profunctor.Composition` からの同等の Haskell 定義です、いくつかの名前を変更した後:

```
data Procompose q p a b where
    Procompose :: q a c -> p c b -> Procompose q p a b

trait Procompose[Q[_], _], P[_], _], A, B]

object Procompose{
```

```
def apply[Q[_], _, P[_], _, A, B, C]
  (qac: Q[A, C])(pcb: P[C, B]): Procompose[Q, P, A, B] = ???  
}
```

これは、一般化代数データ型、または GADT 構文を使用しています。ここで自由な型変数 (ここでは `c`) は自動的に存在量化されます。したがって、(非 Curry 化された) データコンストラクタ `Procompose` は次に等価です:

```
exists c. (q a c, p c b)
```

定義された合成の単位はホム関手です。これは忍者米田の補題から直ちに従います。したがって、プロ関手が射として機能する圏があるかどうかを尋ねることは理にかなっています。答えは肯定的ですが、プロ関手の合成の結合性と恒等性の規則は自然同型に関してのみ成り立つという注意が必要です。このような規則が同型によってのみ成り立つ圏は、双圏 (それは 2-圏よりも一般的です) と呼ばれます。したがって、双圏 `Prof` があります。ここで、対象は圏であり、射はプロ関手であり、射間の射 (二セルとも呼ばれます) は自然変換です。実際、さらに進んで、通常の関手と共にプロ関手も圏間の射として持つことができます。二つの射のタイプを持つ圏は二重圏と呼ばれます。

プロ関手は、Haskell の `lens` ライブラリや `arrow` ライブラリで重要な役割を果たします。

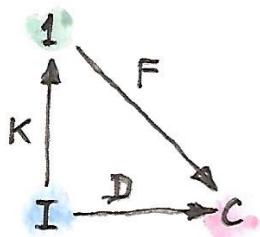
27

Kan 拡張

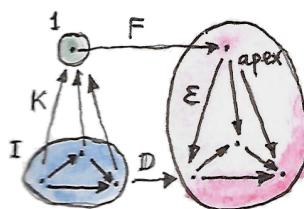
これまでのところ、私たちは主に単一の圏や一組の圏と一緒に作業をしてきました。場合によっては、これが少し制約となることもあります。

例えば、圏 C で極限を定義する際には、私たちはその錐の形成の基礎となるパターンのテンプレートとして索引圏 I を導入しました。錐の頂点のテンプレートとして別の圏、たとえば単純なものを導入することも理にかなっていました。代わりに、私たちは I から C への定数関手 Δ_c を使いました。

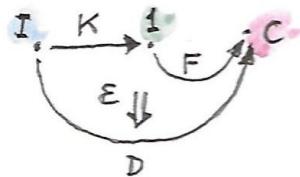
この不格好さを修正する時が来ました。3つの圏を使って極限を定義しましょう。まず、索引圏 I から圏 C への関手 D から始めます。これは錐の基盤となる図式を選ぶ関手、つまり図式関手です。



新しい追加は、单一の対象 (そして単一の恒等射) を含む圏 1 です。 I からこの圏への可能な関手 K は一つだけです。それはすべての対象を 1 の唯一の対象にマッピングし、すべての射を恒等射にマッピングします。任意の関手 F から 1 への C は私たちの錐のための潜在的な頂点を選びます。

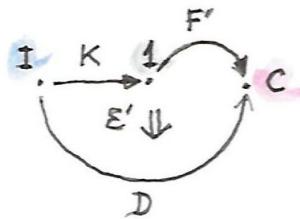


錐は、 $F \circ K$ から D への自然変換 ε です。 $F \circ K$ は私たちの元の Δ_c と全く同じことをします。次の図はこの変換を示しています。



今、私たちは「最良」な関手 F を選ぶ普遍的な性質を定義できます。この F は 1 を D の極限である対象にマッピングし、 $F \circ K$ から D への自然変換 ε は対応する射影を提供します。この普遍的な関手は、 D に沿った K の右 Kan 拡張と呼ばれ、 $\mathbf{Ran}_K D$ と表記されます。

普遍的な性質を定式化しましょう。別の錐、つまり別の関手 F' と、 $F' \circ K$ から D への自然変換 ε' を持っていると仮定します。

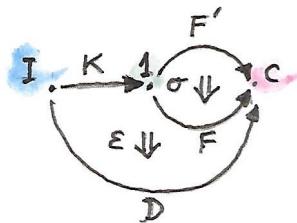


もし Kan 拡張 $F = \mathbf{Ran}_K D$ が存在するならば、 F' からそれに向けての唯一の自然変換 σ が存在し、 ε' は ε を通じて因数分解されることが求

められます。つまり:

$$\varepsilon' = \varepsilon \cdot (\sigma \circ K)$$

ここで、 $\sigma \circ K$ は 2 つの自然変換 (そのうちの 1 つは K 上の恒等自然変換) の水平合成です。この変換はその後 ε と垂直に合成されます。



コンポーネントでは、 I の対象 i に作用するとき、私たちは次の式を得ます:

$$\varepsilon'_i = \varepsilon_i \circ \sigma_{Ki}$$

私たちの場合、 σ は 1 の単一の対象に対応する唯一のコンポーネントを持っています。だから、これは実際には F' によって定義された錐の頂点から $\mathbf{Ran}_K D$ によって定義された普遍錐の頂点への唯一の射です。交換条件は極限の定義によって要求されるものです。

しかし、重要なのは、私たちは単純な圈 1 を任意の圈 A に置き換える自由があり、右 Kan 拡張の定義は有効のままであるということです。

27.1 右 Kan 拡張

索引圏 I から圏 C への関手 $D :: I \rightarrow C$ に沿った関手 $K :: I \rightarrow A$ の右 Kan 拡張は、関手 $F :: A \rightarrow C$ ($\text{Ran}_K D$ と表記されます) と自然変換

$$\varepsilon :: F \circ K \rightarrow D$$

の組み合わせです。その他の任意の関手 $F' :: A \rightarrow C$ と自然変換

$$\varepsilon' :: F' \circ K \rightarrow D$$

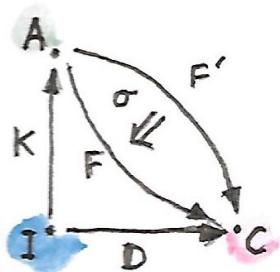
が与えられた場合には、 ε' を因数分解する唯一の自然変換

$$\sigma :: F' \rightarrow F$$

が存在します:

$$\varepsilon' = \varepsilon \cdot (\sigma \circ K)$$

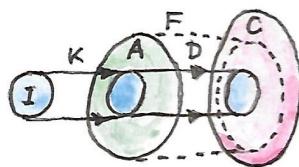
これはかなり難解ですが、次の素敵な図で視覚化できます:



これを見ると、ある意味で、Kan 拡張は「関手の乗算の逆」として作用することに気づけます。一部の著者は、 $\mathbf{Ran}_K D$ のために D/K という表記を使います。確かに、この表記では、 ε の定義も、右 Kan 拡張の余単位とも呼ばれるものも、単純な相殺のように見えます：

$$\varepsilon :: D/K \circ K \rightarrow D$$

Kan 拡張のもう一つの解釈は、関手 K が圏 I を A の内部に埋め込むことを考慮するものです。最も単純なケースでは、 I は A の部分圏かもしれません。私たちは I を C にマッピングする関手 D を持っています。 D を A 全体に定義された関手 F に拡張できますか？理想的には、そのような拡張は $F \circ K$ を D と同型にするでしょう。言い換えれば、 F は D の始域を A に拡張するでしょう。しかし、完全な同型は通常求められるものとしては多すぎるので、私たちはただその半分、つまり $F \circ K$ から D への片方向の自然変換 ε で満足できます。（左 Kan 拡張は別の方向を選びます。）



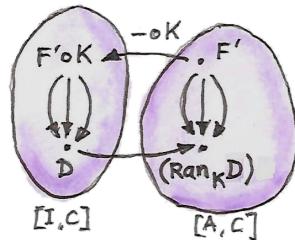
もちろん、関手 K が対象に单射でない場合や射集合に忠実でない場合は、埋め込みの絵が崩れます。その場合、Kan 拡張は失われた情報を最善を尽くして外挿しようとします。

27.2 Kan 拡張と隨伴

今、任意の D (固定された K と共に) に対して右 Kan 拡張が存在すると仮定します。その場合、 $\mathbf{Ran}_K -$ (ダッシュが D を置き換える) は、関手圏 $[\mathbf{I}, \mathbf{C}]$ から関手圏 $[\mathbf{A}, \mathbf{C}]$ への関手です。実は、この関手は前合成関手 $- \circ K$ の右随伴です。後者は $[\mathbf{A}, \mathbf{C}]$ の関手を $[\mathbf{I}, \mathbf{C}]$ の関手にマッピングします。随伴は以下のようになります:

$$[\mathbf{I}, \mathbf{C}](F' \circ K, D) \cong [\mathbf{A}, \mathbf{C}](F', \mathbf{Ran}_K D)$$

これは私たちが ε' と呼んだすべての自然変換に対応する唯一の自然変換が σ であるという事実の単なる再表現です。



さらに、私たちが圏 \mathbf{I} を \mathbf{C} と同じものと選んだ場合、 D の代わりに恒等関手 $I_{\mathbf{C}}$ を置き換えることができます。そうすると、次の等式が得られます:

$$[\mathbf{C}, \mathbf{C}](F' \circ K, I_{\mathbf{C}}) \cong [\mathbf{A}, \mathbf{C}](F', \mathbf{Ran}_K I_{\mathbf{C}})$$

今度は、 F' を $\mathbf{Ran}_K I_C$ と同じものとして選べば、右側に恒等自然変換が含まれ、それに対応して左側には次の自然変換が得られます：

$$\varepsilon :: \mathbf{Ran}_K I_C \circ K \rightarrow I_C$$

これは随伴の余単位とよく似ています：

$$\mathbf{Ran}_K I_C \dashv K$$

実際に、関手 K に沿って恒等関手の右 Kan 拡張を使用すると、 K の左随伴を計算するために使用できます。そのためには、もう一つの条件が必要です：右 Kan 拡張は関手 K によって保存されなければなりません。拡張の保存とは、 K で事前合成された関手の Kan 拡張を計算した場合、それが K で事前合成された元の Kan 拡張と同じ結果になることを意味します。私たちの場合、この条件は次のように単純化されます：

$$K \circ \mathbf{Ran}_K I_C \cong \mathbf{Ran}_K K$$

注意すべきは、 K による除算の表記を使用すると、随伴は次のように書けるということです：

$$I/K \dashv K$$

これは、随伴が何らかの逆を記述することを確認してくれます。保存条件は次のようになります：

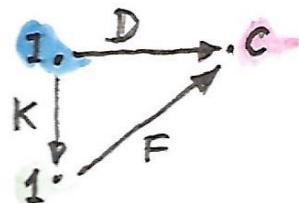
$$K \circ I/K \cong K/K$$

関手に沿った関手自体の右 Kan 拡張、 K/K 、は密度モナドと呼ばれます。

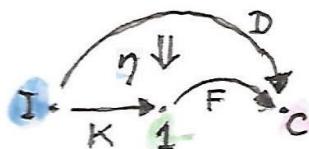
随伴の公式は重要な結果であり、私たちはすぐに見るように、エンド(余エンド)を使用して Kan 拡張を計算できるため、実用的な方法を提供してくれます。

27.3 左 Kan 拡張

左 Kan 拡張は、右 Kan 拡張の双対構造です。いくつかの直感を得るために、まず単一の圏 $\mathbf{1}$ を使用して余極限の定義を構造化してみましょう。私たちは、関手 $D : \mathbf{I} \rightarrow \mathbf{C}$ を使用してその基盤を形成し、関手 $F : \mathbf{1} \rightarrow \mathbf{C}$ を使用してその頂点を選択します。

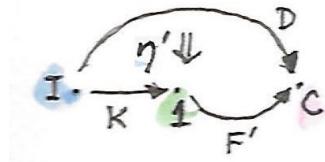


余錐の側面、つまり射入れは、 D から $F \circ K$ への自然変換 η のコンポーネントです。

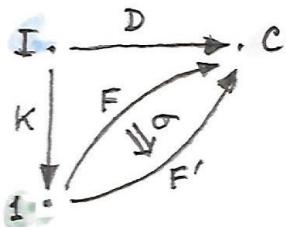


余極限は普遍余錐です。つまり、他の任意の関手 F' と自然変換

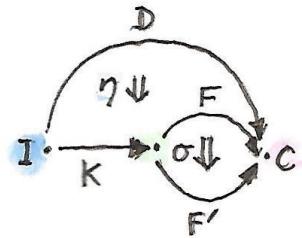
$$\eta' : D \rightarrow F' \circ K$$



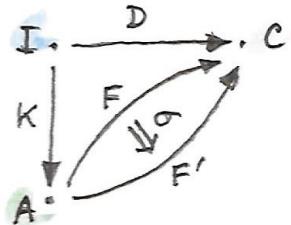
に対して、 F から F' への唯一の自然変換 σ が存在します。



そのようにして、次の図で示されるようになります:



单一の圏 \mathbf{I} を \mathbf{A} に置き換えると、この定義は自然に左 Kan 拡張の定義に一般化されます。これは $\mathbf{Lan}_K D$ と表記されます。



自然変換:

$$\eta : D \rightarrow \mathbf{Lan}_K D \circ K$$

は左 Kan 拡張の単位と呼ばれます。

以前のように、自然変換間の一対一の対応を随伴の観点から述べ直すことができます:

$$[\mathbf{A}, \mathbf{C}](\mathbf{Lan}_K D, F') \cong [\mathbf{I}, \mathbf{C}](D, F' \circ K)$$

言い換えると、左 Kan 拡張は前合成と K の左随伴であり、右 Kan 拡張は右随伴です。

恒等関手の右 Kan 拡張を使用して K の左随伴を計算できるのと同様に、恒等関手の左 Kan 拡張は K の右随伴になります (η は随伴の単位です):

$$K \dashv \mathbf{Lan}_K I_{\mathbf{C}}$$

この二つの結果を組み合わせると、次が得られます:

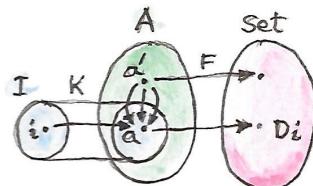
$$\mathbf{Ran}_K I_C \dashv K \dashv \mathbf{Lan}_K I_C$$

27.4 エンドとしての Kan 拡張

Kan 拡張の真の力は、エンドと余エンドを使用して計算できるという事実から来ています。単純化のために、私たちはターゲットの圏 C を \mathbf{Set} として考えますが、公式は任意の圏に拡張することができます。

Kan 拡張を使用して関手の作用をその元の始域の外側に拡張するというアイデアを再考してみましょう。 K が I を A の内部に埋め込むと仮定します。関手 D は I を \mathbf{Set} にマッピングします。 K の像内の任意の対象 a 、つまり $a = Ki$ に対して、拡張された関手は a を Di にマッピングすると単に言えばよいでしょう。しかし、 K の像の外側にある A の対象はどうでしょうか？ 各対象は潜在的に K の像のすべての対象に多くの射を介して接続されています。関手はこれらの射を保存しなければなりません。対象 a から K の像への射の全体は、ホム関手によって特徴づけられます:

$$A(a, K-)$$



このホム関手は 2 つの関手の合成です:

$$\mathbf{A}(a, K-) = \mathbf{A}(a, -) \circ K$$

右 Kan 拡張は関手合成の右随伴です:

$$[\mathbf{I}, \mathbf{Set}](F' \circ K, D) \cong [\mathbf{A}, \mathbf{Set}](F', \mathbf{Ran}_K D)$$

ホム関手を F' に置き換えるとどうなるか見てみましょう:

$$[\mathbf{I}, \mathbf{Set}](\mathbf{A}(a, -) \circ K, D) \cong [\mathbf{A}, \mathbf{Set}](\mathbf{A}(a, -), \mathbf{Ran}_K D)$$

そして、合成をインライン展開します:

$$[\mathbf{I}, \mathbf{Set}](\mathbf{A}(a, K-), D) \cong [\mathbf{A}, \mathbf{Set}](\mathbf{A}(a, -), \mathbf{Ran}_K D)$$

右側は米田の補題を使用して簡略化できます:

$$[\mathbf{I}, \mathbf{Set}](\mathbf{A}(a, K-), D) \cong \mathbf{Ran}_K D a$$

これで、右 Kan 拡張の非常に便利な式を自然変換の集合としてエンドを書き換えることができます:

$$\mathbf{Ran}_K D a \cong \int_i \mathbf{Set}(\mathbf{A}(a, Ki), Di)$$

左 Kan 拡張に対しても、余エンドを用いた類似の公式があります:

$$\mathbf{Lan}_K D a = \int^i \mathbf{A}(Ki, a) \times Di$$

これが実際に関手合成の左随伴であることを示すために、左側の式を代入して見ましょう：

$$[\mathbf{A}, \mathbf{Set}](\mathbf{Lan}_K D, F') \cong [\mathbf{I}, \mathbf{Set}](D, F' \circ K)$$

左側を置換して：

$$[\mathbf{A}, \mathbf{Set}]\left(\int^i \mathbf{A}(Ki, -) \times Di, F'\right)$$

これは自然変換の集合なので、エンドとして書き直せます：

$$\int_a \mathbf{Set}\left(\int^i \mathbf{A}(Ki, a) \times Di, F'a\right)$$

ホム関手の連続性を使用して、余エンドをエンドと置き換えます：

$$\int_a \int_i \mathbf{Set}(\mathbf{A}(Ki, a) \times Di, F'a)$$

積-指数随伴を使用します：

$$\int_a \int_i \mathbf{Set}(\mathbf{A}(Ki, a), (F'a)^{Di})$$

指数は対応するホム集合と同型です：

$$\int_a \int_i \mathbf{Set}(\mathbf{A}(Ki, a), \mathbf{A}(Di, F'a))$$

Fubini の定理を使用して、2つのエンドを交換できます：

$$\int_i \int_a \mathbf{Set}(\mathbf{A}(Ki, a), \mathbf{A}(Di, F'a))$$

内側のエンドは 2 つの関手間の自然変換の集合を表しているので、米田の補題を使用できます:

$$\int_i \mathbf{A}(Di, F'(Ki))$$

これは、随伴の証明をしようとした右側の自然変換の集合です:

$$[\mathbf{I}, \mathbf{Set}](D, F' \circ K)$$

エンド、余エンド、米田の補題を使用するこの種の計算は、「エンドの計算」の典型的なものです。

27.5 Haskell での Kan 拡張

Kan 拡張のエンド/余エンドの式は、Haskell に簡単に翻訳できます。右拡張から始めましょう:

$$\mathbf{Ran}_K Da \cong \int_i \mathbf{Set}(\mathbf{A}(a, Ki), Di)$$

この定義を見ると、`Ran` は関数が適用される型 `a` の値を含むべきであり、関手 `k` と `d` の間の自然変換も必要です。例えば、`k` が木関手で、`d` がリスト関手であるとし、`Ran Tree [] String` を与えられた場合、関数を渡すと:

```
f :: String -> Tree Int
```

```
def f: String => Tree[Int]
```

`Int` のリストが返されます。右 Kan 拡張はあなたの関数を使って木を生成し、それをリストに再梱包します。例えば、文字列からパース木を生成するパーサーを渡すと、その木の深さ優先探索に対応するリストが得られます。

右 Kan 拡張は、`d` を恒等関手に置き換えることで、与えられた関手の左随伴を計算するために使用することができます。これにより、関手 `k` の左随伴が、次の型の多相的関数の集合で表されます：

```
forall i. (a -> k i) -> i
```

```
type Id[I] = I

trait PolyFunc[A, K[_]] {
    type AtoK[I] = A => K[I]

    def apply(): AtoK ~> Id
}
```

例えば、`k` がモノイドの圏から忘却関手であると仮定します。すると、全称量化子はすべてのモノイドを渡ります。もちろん、Haskell ではモノイド則を表現することはできませんが、以下は結果として得られる自由関手（忘却関手 `k` は対象に対して恒等的です）のまとまな近似です：

```
type Lst a = forall i. Monoid i => (a -> i) -> i

trait `PolyFunctionM`[F[_], G[_]] {
  def apply[I: Monoid](fa: F[I]): G[I]
}

trait Lst[A] {
  type aTo[X] = A => X

  def apply(): aTo `PolyFunctionM` Id
}
```

予想通り、これは自由モノイド、つまり Haskell のリストを生成します:

```
toLst :: [a] -> Lst a
toLst as = \f -> foldMap f as

fromLst :: Lst a -> [a]
fromLst f = f (\a -> [a])

// Read more about foldMap:
// typelevel.org/cats/typeclasses/foldable.html
def foldMap[F[_], A, B](fa: F[A])(f: A => B)
  (implicit B: Monoid[B]): B = ???
implicit def listMonoid[A]: Monoid[List[A]] = ???
```

```

def toLst[A]: List[A] => Lst[A] = as => new Lst[A] {
  def apply(): `PolyFunctionM`[aTo, Id] =
    new `PolyFunctionM`[aTo, Id] {
      def apply[I: Monoid](fa: aTo[I]): Id[I] =
        foldMap(as)(fa)
    }
}

def fromLst[A]: Lst[A] => List[A] =
  f => f().apply(a => List(a))

```

左 Kan 拡張は余エンドです:

$$\mathbf{Lan}_K Da = \int^i A(Ki, a) \times Di$$

したがって、存在量化子として表されます。記号的には:

```
Lan k d a = exists i. (k i -> a, d i)
```

これは、GADT を使用するか、または全称量化子を持つデータコンストラクタを使用して、Haskell でエンコードできます:

```

data Lan k d a = forall i. Lan (k i -> a) (d i)

trait Lan[K[_], +D[_], A] {
  def fk[I](ki: K[I]): A
  def di[I]: D[I]
}

```

このデータ構造の解釈は、それが未指定の i のコンテナを取り、 a を生成する関数を含んでいるというものです。また、それらの i のコンテナも持っています。どのような i であるかわからないので、このデータ構造でできる唯一のこととは、 i のコンテナを取り出し、自然変換を使用して関手 k で定義されたコンテナに再梱包し、関数を呼び出して a を得ることです。例えば、 d が木であり、 k がリストであれば、木をシリアル化し、結果のリストで関数を呼び出し、 a を得ることができます。

左 Kan 拡張は、関手の右随伴を計算するために使用することができます。積関手の右随伴が指數関数であることがわかっているので、Kan 拡張を使用して実装してみましょう：

```
type Exp a b = Lan ((,) a) I b

type Exp[A, B] = Lan[(A, ?), I, B]
```

これは関数型と同型であることが、以下の一对の関数によって証明されます：

```
toExp :: (a -> b) -> Exp a b
toExp f = Lan (f . fst) (I ())

fromExp :: Exp a b -> (a -> b)
fromExp (Lan f (I x)) = \a -> f (a, x)
```

```

def fst[I]: ((I, _)) => I = _._1

def toExp[A, B]: (A => B) => Exp[A, B] = f => new Lan[(A, ?), I, B] {
  def fk[L](ki: (A, L)): B =
    f.compose(fst[A])(ki)

  def di[L]: I[L] = I()
}

def fromExp[A, B]: Exp[A, B] => (A => B) =
  lan => a => lan.fk((a, lan.di))

```

以前に一般的なケースで説明したように、私たちは次のステップを実行しました:

1. x のコンテナを取り出し (ここでは、単なる自明な恒等コンテナ)、関数 f を取り出しました。
2. 自然変換を使用してコンテナを恒等関手からペア関手へと再梱包しました。
3. 関数 f を呼び出しました。

27.6 自由関手

Kan 拡張の興味深い応用の一つは自由関手の構成です。これは、次のような実用的な問題の解決策です: 型コンストラクタ、つまり対象

のマッピングを持っている場合、この型コンストラクタに基づいて関手を定義することは可能でしょうか？ 言い換えれば、この型コンストラクタを完全な自己関手に拡張するための射のマッピングを定義できますか？

重要な観察は、型コンストラクタが離散圏の関手として記述できるということです。離散圏は恒等射以外の射を持たない圏です。与えられた圏 C に対して、私たちはすべての非恒等射を単純に捨てることによって離散圏 $|C|$ を常に構成できます。 $|C|$ から C への関手 F は、その後、対象の単純なマッピング、つまり Haskell で言うところの型コンストラクタです。また、 $|C|$ を C に注入する標準的な関手 J もあります。それは対象に対して（そして恒等射に対しても）恒等的です。 F の J に沿った左 Kan 拡張が存在する場合、それは C から C への関手です：

$$\text{Lan}_J Fa = \int^i C(Ji, a) \times Fi$$

これは、 F に基づく自由関手と呼ばれます。

Haskell では、それを次のように書くでしょう：

```
data FreeF f a = forall i. FMap (i -> a) (f i)

trait FreeF[F[_], A] {
  def h[I]: I => A
  def fi[I]: F[I]
}
```

実際には、任意の型コンストラクタ f に対して、 $\text{FreeF } f$ は関手です：

```
instance Functor (FreeF f) where
    fmap g (FMap h fi) = FMap (g . h) fi

implicit def freeFunctor[F[_]] = new Functor[FreeF[F, ?]] {
    def fmap[A, B](g: A => B)(fa: FreeF[F, A]): FreeF[F, B] = {
        new FreeF[F, B] {
            def h[I]: I => B = g compose fa.h
            def fi[I]: F[I] = fi
        }
    }
}
```

ご覧のとおり、自由関手は関数の持ち上げを、その関数とその引数の両方を記録することによって偽装します。それは関数の合成を記録することによって持ち上げられた関数を蓄積します。関手の規則は自動的に満たされます。この構成は、論文 [Freer Monads, More Extensible Effects^{*1}](#) で使用されました。

代わりに、同じ目的のために右 Kan 拡張を使用することもできます：

```
newtype FreeF f a = FreeF (forall i. (a -> i) -> f i)
```

^{*1} <http://okmij.org/ftp/Haskell/extensible/more.pdf>

```
case class FreeF[F[_], A](r: (A => ?) ~> F)
```

これが実際に関手であることは簡単に確認できます:

```
instance Functor (FreeF f) where
  fmap g (FreeF r) = FreeF (\bi -> r (bi . g))

implicit def freeFunctor[F[_]] = new Functor[FreeF[F, ?]] {
  def fmap[A, B](g: A => B)(fa: FreeF[F, A]): FreeF[F, B] = fa match {
    case FreeF(r) => FreeF {
      new ~>[B => ?, F] {
        def apply[C](bi: B => C): F[C] =
          r(bi compose g)
      }
    }
  }
}
```

28

豊穣圏

圏はその対象が集合を形成するとき小さいとされます。しかし、集合より大きなものが存在することが知られています。有名な例として、全ての集合の集合は標準的な集合論 (Zermelo-Fraenkel 理論、必要に応じて選択公理を付加したもの) 内では形成できません。従って、全ての集合の圏は大きなものでなければなりません。Grothendieck 宇宙のような数学的トリックを使って、集合を超えたコレクションを定義することができます。これらのトリックにより、大きな圏について語ることができます。

圏が局所的に小さいとは、任意の 2 つの対象間の射が集合を形成する場合を指します。もし集合を形成しない場合は、いくつかの定義を

再考する必要があります。特に、射を集合から選び出すことができない場合、射の合成をどのように意味するかです。解決策として、対象を集合圏 **Set** の対象ではなく、別の圏 **V** の対象に置き換えることによって、ホム集合をブートストラップします。一般に、対象は要素を持たないため、個々の射について話すことはもはや許されません。ホム対象全体に対して実行できる操作の観点から、**豊穣圏**のすべての性質を定義する必要があります。これを行うためには、ホム対象を提供する圏には追加の構造が必要です – それはモノイダル圏でなければなりません。このモノイダル圏を **V** と呼ぶとすると、**V** に豊穣化された圏 **C** について語ることができます。

大きさの理由以外に、単なる集合よりも多くの構造を持つものにホム集合を一般化することに興味があるかもしれません。例えば、伝統的な圏は対象間の距離の概念を持っていません。対象は射によって接続されているか、そうでないかのどちらかです。ある対象に接続されている全ての対象はその近隣です。現実とは異なり、圏では友達の友達の友達は自分の親友と同じくらい近いです。適切に豊穣化された圏では、対象間の距離を定義することができます。

豊穣圏についての経験を積むためのもう一つの非常に実用的な理由は、圏論的知識の非常に便利なオンラインソースである **nLab**^{*1}が、主に豊穣圏の観点から書かれているためです。

^{*1} <https://ncatlab.org/>

28.1 なぜモノイダル圏か？

豊穣圏を構成する際には、モノイダル圏を **Set** と置き換え、ホム対象をホム集合と置き換えたときに、通常の定義を回復できるように心に留めておく必要があります。これを達成する最善の方法は、通常の定義から始め、点を使わない方法(つまり、集合の要素を名付けることなく)でそれらを再定式化することです。

まず合成の定義から始めましょう。通常、合成はペアの射を取ります。一つは $\mathbf{C}(b, c)$ から、もう一つは $\mathbf{C}(a, b)$ からで、これを $\mathbf{C}(a, c)$ からの射にマップします。言い換えると、それは次のようなマッピングです:

$$\mathbf{C}(b, c) \times \mathbf{C}(a, b) \rightarrow \mathbf{C}(a, c)$$

これは集合間の関数です – そのうちの一つは 2 つのホム集合のデカルト積です。この公式は、デカルト積をもっと一般的なものに置き換えることで簡単に一般化できます。圏論的積が機能しますが、さらに一般的なテンソル積を使用することもできます。

次に恒等射についてです。ホム集合から個々の要素を選ぶ代わりに、単集合 **1** からの関数を使ってそれらを定義することができます:

$$j_a :: \mathbf{1} \rightarrow \mathbf{C}(a, a)$$

再び、単集合を終対象に置き換えることができますが、テンソル積の単位要素 i に置き換えることでさらに進むことができます。

ご覧のとおり、あるモノイダル圏 \mathbf{V} から取られた対象はホム集合の置き換えに適しています。

28.2 モノイダル圏

モノイダル圏について以前話しましたが、定義を再度述べる価値があります。モノイダル圏はテンソル積を定義し、それは双関手です：

$$\otimes :: \mathbf{V} \times \mathbf{V} \rightarrow \mathbf{V}$$

テンソル積が結合的であることを望んでいますが、自然同型まで満たせば十分です。この同型は結合子 (associator) と呼ばれ、そのコンポーネントは次のとおりです：

$$\alpha_{abc} :: (a \otimes b) \otimes c \rightarrow a \otimes (b \otimes c)$$

これは全ての 3 つの引数において自然でなければなりません。

モノイダル圏はまた、特別な単位対象 i を定義しなければなりません。それはテンソル積の単位要素として機能し、再び自然同型までです。2 つの同型はそれぞれ左単位子 (left uniter) と右単位子 (right uniter) と呼ばれ、それらのコンポーネントは以下の通りです：

$$\lambda_a :: i \otimes a \rightarrow a$$

$$\rho_a :: a \otimes i \rightarrow a$$

結合子と単位子は整合性条件を満たさなければなりません：

$$\begin{array}{ccc}
 ((a \otimes b) \otimes c) \otimes d & \xrightarrow{\alpha_{abc} \otimes \text{id}_d} & (a \otimes (b \otimes c)) \otimes d \\
 \downarrow \alpha_{(a \otimes b)cd} & & \downarrow \alpha_{a(b \otimes c)d} \\
 (a \otimes b) \otimes (c \otimes d) & & a \otimes ((b \otimes c) \otimes d) \\
 & \searrow \alpha_{ab(c \otimes d)} & \swarrow \text{id}_a \otimes \alpha_{bcd} \\
 & a \otimes (b \otimes (c \otimes d)) &
 \end{array}$$

$$\begin{array}{ccc}
 (a \otimes i) \otimes b & \xrightarrow{\alpha_{aib}} & a \otimes (i \otimes b) \\
 & \searrow \rho_a \otimes \text{id}_b & \swarrow \text{id}_a \otimes \lambda_b \\
 & a \otimes b &
 \end{array}$$

モノイダル圏が対称的であるとは、次のような自然同型のコンポーネントがあることを意味します:

$$\gamma_{ab} :: a \otimes b \rightarrow b \otimes a$$

その「二乗が 1」です:

$$\gamma_{ba} \circ \gamma_{ab} = \text{id}_{a \otimes b}$$

そしてそれがモノイダル構造と整合している必要があります。

モノイダル圏についての興味深い点は、内部ホム (関数対象) をテンソル積の右随伴として定義できるかもしれないことです。覚えておくべきことは、関数対象、または指数は、圏論的積の右随伴を通して標

準的に定義されました。そのような対象が任意の対象のペアに対して存在した圏はデカルト閉と呼ばれました。ここではモノイダル圏における内部ホムの随伴を定義します:

$$\mathbf{V}(a \otimes b, c) \sim \mathbf{V}(a, [b, c])$$

以下に従って、G. M. Kelly^{*2}の記法を使います。内部ホムに $[b, c]$ を使用しています。この随伴の余単位は評価射と呼ばれる自然変換のコンポーネントです:

$$\varepsilon_{ab} :: ([a, b] \otimes a) \rightarrow b$$

テンソル積が対称的でない場合、次の随伴を使用して別の内部ホムを定義することができます:

$$\mathbf{V}(a \otimes b, c) \sim \mathbf{V}(b, [[a, c]])$$

両方が定義されているモノイダル圏は双閉と呼ばれます。Set における自己関手の圏は、関手合成をテンソル積として使用する、双閉ではない圏の例です。これはモナドを定義するために使用した圏です。

28.3 豊穣圏

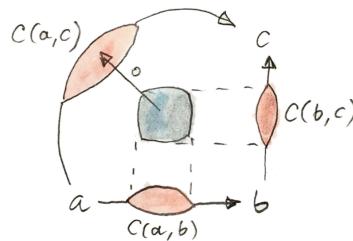
モノイダル圏 \mathbf{V} に豊穣化された圏 \mathbf{C} はホム集合をホム対象に置き換えます。 \mathbf{C} の任意の対象ペア a と b に対して、 \mathbf{V} の対象 $\mathbf{C}(a, b)$ を関連付けます。ホム対象にも同じ記法を使用しますが、それが射を含ま

^{*2} <http://www.tac.mta.ca/tac/reprints/articles/10/tr10.pdf>

ないことを理解しています。一方で \mathbf{V} は通常の（豊穣化されていない）圏で、ホム集合と射を持っています。従って、集合から完全には脱却していません — ただそれを隠しています。

\mathbf{C} の個々の射について話すことができないため、射の合成は \mathbf{V} の射の族に置き換えられます：

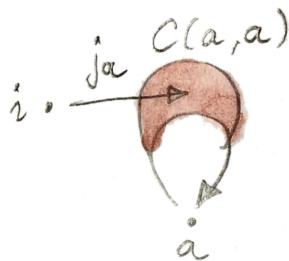
$$\circ :: \mathbf{C}(b, c) \otimes \mathbf{C}(a, b) \rightarrow \mathbf{C}(a, c)$$



同様に、恒等射は \mathbf{V} の射の族に置き換えられます：

$$j_a :: i \rightarrow \mathbf{C}(a, a)$$

ここで i は \mathbf{V} のテンソル単位です。



合成の結合性は \mathbf{V} の結合子に関して定義されます:

$$\begin{array}{ccc}
 (\mathbf{C}(c, d) \otimes \mathbf{C}(b, c)) \otimes \mathbf{C}(a, b) & \xrightarrow{\circ \otimes \text{id}} & \mathbf{C}(b, d) \otimes \mathbf{C}(a, b) \\
 \downarrow \alpha & & \searrow \circ & \nearrow \circ \\
 \mathbf{C}(c, d) \otimes (\mathbf{C}(b, c) \otimes \mathbf{C}(a, b)) & \xrightarrow{\text{id} \otimes \circ} & \mathbf{C}(c, d) \otimes \mathbf{C}(a, c) & \mathbf{C}(a, d)
 \end{array}$$

単位則は同様に単位子を用いて表されます:

$$\begin{array}{ccc}
 C(a,b) \otimes i & \xrightarrow{\text{id} \otimes j_a} & C(a,b) \otimes C(a,a) \\
 \rho \searrow & & \swarrow \circ \\
 & C(a,b) & \\
 i \otimes C(a,b) & \xrightarrow{j_b \otimes \text{id}} & C(b,b) \otimes C(a,b) \\
 \lambda \searrow & & \swarrow \circ \\
 & C(a,b) &
 \end{array}$$

28.4 前順序

前順序は薄い圏、つまり各ホム集合が空であるか単集合である圏として定義されます。非空集合 $C(a,b)$ を a が b 以下である証明と解釈します。そのような圏は、非常に単純なモノイダル圏である *False* と *True* (時には 0 と 1 とも呼ばれる) の 2 つの対象だけを含むモノイダル圏に豊穣化することができます。必須の恒等射の他に、この圏には $0 \rightarrow 1$ と行く单一の射があります。それに対して単純なモノイダル構造が確立されており、テンソル積は 0 と 1 の単純な算術をモデリングします (つまり、唯一の非ゼロ積は $1 \otimes 1$ です)。この圏の恒等対象は 1 です。これは厳密なモノイダル圏であり、結合子と単位子は恒等射です。

前順序において、ホム集合は空または単集合なので、私たちはそれを容易に我々の小さな圏からのホム対象に置き換えることができます。

す。豊穣化された前順序 \mathbf{C} は、任意の対象ペア a と b に対するホム対象 $\mathbf{C}(a, b)$ を持ちます。もし a が b 以下ならば、この対象は 1 です。そうでなければ 0 です。

合成について見てみましょう。任意の 2 つの対象のテンソル積は 0 ですが、両方が 1 の場合に限り 1 です。それが 0 の場合、合成射には 2 つの選択肢があります。 \mathbf{id}_0 か $0 \rightarrow 1$ のどちらかです。しかし、それが 1 の場合、唯一の選択肢は \mathbf{id}_1 です。これを関係に戻して解釈すると、 $a \leq b$ かつ $b \leq c$ ならば $a \leq c$ です。これはまさに必要な推移則です。

恒等射についてはどうでしょうか。それは 1 から $\mathbf{C}(a, a)$ への射です。1 から行く唯一の射は恒等射 \mathbf{id}_1 なので、 $\mathbf{C}(a, a)$ は 1 でなければなりません。これは $a \leq a$ であり、前順序の反射則です。従って、推移性と反射性は、前順序を豊穣圏として実装することで自動的に強制されます。

28.5 距離空間

興味深い例の一つは William Lawvere^{*3}によるものです。彼は距離空間が豊穣圏を使って定義され得ることに気づきました。距離空間は任意の 2 つの対象間の距離を定義します。この距離は非負の実数で

^{*3} <http://www.tac.mta.ca/tac/reprints/articles/1/tr1.pdf>

す。無限大を可能な値として含めるのが便利です。距離が無限大の場合、始点の対象から目的の対象へ到達する方法はありません。

距離によって満たされなければならないいくつかの明白な性質があります。その一つは、対象からそれ自身への距離がゼロでなければならぬことです。もう一つは三角不等式です。直接の距離は中間停止点での距離の和を超えることはありません。距離が対称的でないことを要求しないのは最初は奇妙に思えるかもしれません、Lawvere が説明したように、一方の方向では上り坂を歩いている間に、他方の方向では下り坂を歩いていると想像することができます。いずれにせよ、対称性は後で追加の制約として課されるかもしれません。

では、どのようにして距離空間を圏論の言葉に落とし込むことができるのでしょうか？ ホム対象が距離である圏を構成する必要があります。注意してください、距離は射ではなくホム対象です。ホム対象が数である場合にのみ、これらの数が対象であるモノイダル圏 V を構成することができます。非負の実数（プラス無限）は全順序を形成するので、薄い圏として扱うことができます。2つの数 x と y の間の射は $x \geq y$ の場合にのみ存在します（注：これは前順序の定義で通常使用される方向とは逆です）。モノイダル構造は加算によって与えられ、ゼロが単位対象として機能します。言い換えると、2つの数のテンソル積はそれらの和です。

距離空間はそのようなモノイダル圏上で豊穣化された圏です。対象 a から b へのホム対象 $C(a, b)$ は、非負の（無限大の可能性もある）数で

あり、それを a から b への距離と呼びます。恒等射と合成がそのような圏でどのように機能するか見てみましょう。

定義によれば、テンソル単位である数ゼロからホム対象 $C(a, a)$ への射は関係です:

$$0 \geq C(a, a)$$

$C(a, a)$ が非負の数であるため、この条件は a から a への距離が常にゼロであることを私たちに伝えます。チェック！

では合成について話しましょう。2つの隣接するホム対象 $C(b, c) \otimes C(a, b)$ から始めます。テンソル積は2つの距離の和として定義されました。合成は V 内のこの積から $C(a, c)$ への射です。 V 内の射は大なりイコール関係として定義されます。言い換えると、 a から b への距離と b から c への距離の和は、 a から c への距離よりも大きいか等しいです。しかし、それは標準的な三角不等式です。チェック！

豊穣化された圏の概念を用いて距離空間を再構成することで、三角不等式と自己距離ゼロの性質を「無料で」得ることができます。

28.6 豊穣関手

関手の定義には射の写像が含まれます。豊穣設定では、個々の射の概念がないため、ホム対象を一括で扱う必要があります。ホム対象はモノイダル圏 V の対象であり、それらの間には射があります。従って、同じモノイダル圏 V に豊穣化された圏間で豊穣関手を定義すること

とは理にかなっています。 \mathbf{V} の射を使用して、2つの豊穣圏間のホム対象を写像することができます。

豊穣関手 F は、2つの圏 \mathbf{C} と \mathbf{D} の間で、対象から対象への写像に加えて、 \mathbf{C} 内の各対象ペアに対して \mathbf{V} の射を割り当てます:

$$F_{ab} :: \mathbf{C}(a, b) \rightarrow \mathbf{D}(Fa, Fb)$$

関手は構造を保存する写像です。通常の関手では合成と恒等射を保存することが意味されました。豊穣設定では、合成の保存は以下の図が可換であることを意味します:

$$\begin{array}{ccc} \mathbf{C}(b, c) \otimes \mathbf{C}(a, b) & \xrightarrow{\quad \circ \quad} & \mathbf{C}(a, c) \\ \downarrow F_{bc} \otimes F_{ab} & & \downarrow F_{ac} \\ \mathbf{D}(Fb, Fc) \otimes \mathbf{D}(Fa, Fb) & \xrightarrow{\quad \circ \quad} & \mathbf{D}(Fa, Fc) \end{array}$$

恒等射の保存は、「恒等を選ぶ」 \mathbf{V} の射の保存に置き換えられます:

$$\begin{array}{ccc} & i & \\ j_a \swarrow & & \searrow j_{Fa} \\ \mathbf{C}(a, a) & \xrightarrow{F_{aa}} & \mathbf{D}(Fa, Fa) \end{array}$$

28.7 自己豊穣化

閉じた対称モノイダル圏は、内部ホムを使用してホム集合を置き換えることで自己豊穣化することができます（上記の定義を参照）。これを実現するために、内部ホムの合成則を定義する必要があります。言い換えれば、次のシグネチャを持つ射を実装する必要があります：

$$[b, c] \otimes [a, b] \rightarrow [a, c]$$

これは、圏論では通常点を使わない実装を使用することを除けば、他のプログラミングタスクとそれほど変わりません。まず、それが所属すべき集合を指定します。この場合、それはホム集合のメンバーです：

$$V([b, c] \otimes [a, b], [a, c])$$

このホム集合は次に同型です：

$$V(([b, c] \otimes [a, b]) \otimes a, c)$$

私たちがこの新しい集合で射を構成できれば、随伴が私たちを元の集合の射に指し示します。これは合成として使用することができます。私たちが利用可能ないくつかの射を合成することによって、この射を構成します。まず、結合子 $\alpha_{[b, c] \otimes [a, b]} a$ を使用して左側の式を再関連付けします：

$$([b, c] \otimes [a, b]) \otimes a \rightarrow [b, c] \otimes ([a, b] \otimes a)$$

それに続いて随伴の余単位 ε_{ab} を使用します：

$$[b, c] \otimes ([a, b] \otimes a) \rightarrow [b, c] \otimes b$$

そして再び余単位 ε_{bc} を使用して c に到達します。これにより、次の射が構成されます:

$$\varepsilon_{bc} \cdot (\mathbf{id}_{[b,c]} \otimes \varepsilon_{ab}) \cdot \alpha_{[b,c][a,b]a}$$

これはホム集合のメンバーです:

$$V(([b,c] \otimes [a,b]) \otimes a, c)$$

随伴は私たちが探していた合成則を提供します。

同様に、恒等射:

$$j_a :: i \rightarrow [a,a]$$

は次のホム集合のメンバーです:

$$V(i, [a,a])$$

これは随伴を通して次に同型です:

$$V(i \otimes a, a)$$

私たちはこのホム集合が左恒等子 λ_a を含むことを知っています。私たちはそれを随伴の下でのイメージとして j_a として定義することができます。

自己豊穣化の実用的な例は、プログラミング言語の型のプロトタイプとして機能する **Set** 圈です。以前見たように、**Set** はデカルト積に関して閉じたモノイダル圏です。**Set**において、任意の 2 つの集合間のホム集合はそれ自体が集合なので、**Set** の対象です。私たちはそれが指数集合に同型であることを知っているので、外部ホムと内部ホム

は等価です。今、自己豊穣化を通じて、指數集合をホム対象として使用し、デカルト積の指數対象の組み合わせに関して合成を表現することができることも知っています。

28.8 2-圏との関連性

私は以前、圏の圏である **Cat** の文脈で 2-圏について話しました。圏間の射は関手ですが、さらに追加の構造があります: 関手間の自然変換です。2-圏では、対象はしばしばゼロセル、射は 1-セル、射間の射は 2-セルと呼ばれます。**Cat** では、0-セルは圏、1-セルは関手、そして 2-セルは自然変換です。

しかし、2 つの圏間の関手もまた圏を形成することに注意してください。したがって、**Cat** では、私たちは本当にホム集合ではなく **ホム圏**を持っています。実際には、**Set** が **Set** 上で豊穣化された圏として扱うことができるよう、**Cat** は **Cat** 上で豊穣化された圏として扱うことができます。さらに一般的には、任意の圏が **Set** 上で豊穣化された圏として扱われるよう、任意の 2-圏は **Cat** 上で豊穣化されると考えることができます。

29

トポス

私は、私たちがプログラミングから離れて硬派な数学に深く潜っていることを認識しています。しかし、次の大きな革命がプログラミングにもたらすものや、それを理解するために必要な数学の種類は決してわかりません。連続時間を持つ関数型リアクティブプログラミングや、Haskell の型システムを依存型で拡張すること、またはプログラミングにおけるホモトピー型理論の探求など、非常に興味深いアイデアが巡っています。

これまでのところ、私は型を値の集合として気楽に同定してきました。これは厳密に正しいわけではなく、プログラミングでは値を計算し、計算は時間がかかり、極端な場合は終了しないかもしれないとい

う事実を考慮に入れていません。発散する計算は、全てのチューリング完全な言語の一部です。

また、集合論がコンピュータ科学や数学自体の基礎として最適ではないという根本的な理由もあります。良い類推は、集合論が特定のアキテクチャに結びついたアセンブリ言語であるというものです。異なるアキテクチャで数学を実行したい場合は、より一般的なツールを使用する必要があります。

一つの可能性は、集合の代わりに空間を使用することです。空間はより多くの構造を持ち、集合に頼ることなく定義されるかもしれません。空間と通常関連付けられているものはトポロジーであり、連續性のようなものを定義するために必要です。そして、トポロジーへの従来のアプローチは、あなたが推測した通り、集合論を通じています。特に、部分集合の概念はトポロジーにとって中心的です。驚くことはありませんが、圏論者は **Set** 以外の圏にこのアイデアを一般化しました。集合論の代わりとしてちょうど良い特性を持つ圏のタイプは **トポス** (複数形: **トポイ**) と呼ばれ、部分集合の一般化された概念を提供します。

29.1 部分対象分類器

まず、要素ではなく関数を使って部分集合のアイデアを表現しようとしてみましょう。ある集合 a から b への任意の関数 f は、 a の像によって b の部分集合を定義します。しかし、同じ部分集合を定義する

多くの関数があります。もっと具体的にする必要があります。始めるために、多くの要素を 1 つに押し込まない単射一つまり、一つの集合を別の集合に「注入する」単射関数に焦点を当てるかもしれません。有限集合の場合、単射関数を一方の集合の要素から別の集合の要素へと並行する矢印として視覚化することができます。もちろん、最初の集合は 2 番目の集合よりも大きくない限り、または矢印は必然的に収束するでしょう。まだいくつかの曖昧さが残っています: 別の集合 a' と別の単射関数 f' が b に同じ部分集合を選ぶかもしれません。しかし、そのような集合は a と同型であると自分自身に簡単に納得させることができます。この事実を利用して、部分集合をその始域による同型関係によって関連する単射関数の族として定義することができます。より正確には、2 つの単射関数:

$$\begin{aligned} f &:: a \rightarrow b \\ f' &:: a' \rightarrow b \end{aligned}$$

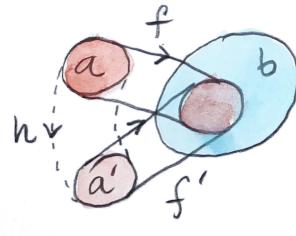
が同型関係であると言われています:

$$h :: a \rightarrow a'$$

そのため:

$$f = f' . h$$

このような同値な単射の族は b の部分集合を定義します。



この定義は、单射関数をモノ射と置き換えることで任意の圏に持ち上げることができます。あなたを思い出させるために、モノ射 m はその普遍的な特性によって定義されます。任意の対象 c と任意の射のペアに対して:

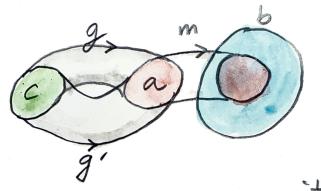
$$g :: c \rightarrow a$$

$$g' :: c \rightarrow a$$

そのような:

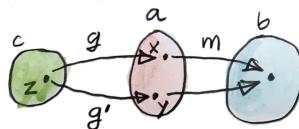
$$m \cdot g = m \cdot g'$$

それは $g = g'$ でなければなりません。



集合上では、この定義は、関数 m がモノ射でない場合、それが何を意味するかを考慮することによって理解しやすいです。それは a の 2 つ

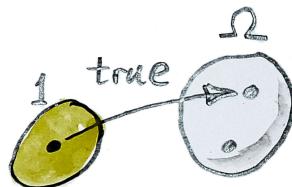
の異なる要素を b の单一要素にマッピングするでしょう。その後、2つの関数 g と g' を見つけることができますが、これらの2つの要素だけが異なります。 m との後方合成はその後、この違いをマスクします。



部分集合を定義する別 の方法は、特性関数と呼ばれる单一の関数を使用することです。これは集合 b から 2 要素集合 Ω への関数 χ です。この集合の1つの要素は「真」として指定され、もう1つは「偽」として指定されます。この関数は、部分集合のメンバーである b の要素に「真」を割り当て、そうでないものに「偽」を割り当てます。

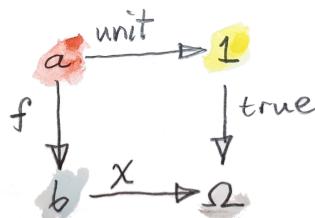
Ω の要素を「真」と指定することの意味を指定する必要が残っています。標準的なトリックを使うことができます: 単集合から Ω への関数を使用します。この関数を *true* と呼びます:

$$\text{true} :: 1 \rightarrow \Omega$$



これらの定義は、部分対象が何であるかだけでなく、要素について話すことなく特別な対象 Ω を定義する方法として組み合わせることができます。アイデアは、「一般的な」部分対象を代表する射 $true$ を望むというものです。**Set** では、2 要素集合 Ω から単一要素の部分集合を選びます。これは可能な限り一般的です。それは明らかに適切な部分集合であり、 Ω にはその部分集合に含まれないもう 1 つの要素があります。

より一般的な設定では、 $true$ を終対象から**分類対象** Ω へのモノ射として定義します。しかし、分類対象を定義する必要があります。この対象と特性関数をリンクする普遍的な特性を必要とします。実際には、**Set** では、特性関数 χ に沿った $true$ の引き戻しは、部分集合 a とそれを b に埋め込む単射関数の両方を定義します。ここに引き戻し図があります：



この図を分析しましょう。引き戻し方程式は：

$$true \cdot unit = \chi \cdot f$$

関数 $true . unit$ は a の各要素を「真」にマップします。したがって f は a の全ての要素を χ が「真」であるところの b の要素にマップしなければなりません。これらは、特性関数 χ によって指定される部分集合の要素です。したがって、 f の像は確かに問題の部分集合です。引き戻しの普遍性は f が単射であることを保証します。

この引き戻し図は、**Set** 以外の圏で分類対象を定義するために使用することができます。そのような圏は終対象を持たなければならず、これによりモノ射 $true$ を定義することができます。それはまた、引き戻しを持たなければなりません – 実際の要件は、それがすべての有限極限を持っていることです (引き戻しは有限極限の一例です)。それらの前提の下で、私たちは分類対象 Ω をその特性によって定義します: すべてのモノ射 f には、引き戻し図を完成させる一意的な射 χ があります。

最後の言明を分析しましょう。引き戻しを構成するとき、私たちは 3 つの対象 Ω 、 b 、そして 1 と 2 つの射、 $true$ と χ を与えられます。引き戻しの存在は、2 つの射 f と $unit$ (後者は終対象の定義によって一意的に決定されます) を備えた最良の対象 a を見つけることができるということを意味します。これにより、図が交通するようにします。

ここでは、異なる方程式系を解いています。 Ω と $true$ を解きながら a と b の両方を変化させます。与えられた a と b には、単射 $f :: a \rightarrow b$ があるかもしれませんし、ないかもしれません。しかし、そうである場合、私たちはそれがいくつかの χ の引き戻しであることを望みま

す。さらに、私たちはこの χ が f によって一意的に決定されることを望んでいます。

单射 f と特性関数 χ の間に一対一の対応関係があるとは言えません、なぜなら引き戻しは同型によってのみ一意であるからです。しかし、以前に部分集合を同値な单射の族として定義したことを思い出してください。私たちは、 b への同値なモノ射の族として b の部分対象を定義することによってそれを一般化することができます。このモノ射の族は、図の引き戻しと同値な族と一対一の対応関係にあります。

したがって、 b の部分対象の集合、 $Sub(b)$ をモノ射の族として定義し、それが Ω への b からの射の集合と同型であることを見ることができます:

$$Sub(b) \cong C(b, \Omega)$$

これは 2 つの関手の自然な同型です。言い換えると、 $Sub(-)$ は Ω という対象の表現可能な(反変)関手です。

29.2 トポス

トポスは以下の特性を持つ圏です:

1. デカルト閉である: すべての積、終対象、および右随伴として定義される指数関数(指数対象)を持っています。
2. すべての有限図表に対して極限を持っています。
3. 部分対象分類器 Ω を持っています。

この属性の集合は、ほとんどのアプリケーションで **Set** に代わるトポスをするのに十分です。それはまた、その定義から導かれる追加の属性を持っています。例えば、トポスは始対象を含むすべての有限余極限を持っています。

部分対象分類器を終対象の 2 つのコピーの余積(和)として定義することは誘惑的です — それが **Set** で何であるかです — しかし、私たちはそれよりも一般的でありたいです。これが真実であるトポスはブル型と呼ばれます。

29.3 トポスと論理

集合論では、特性関数は集合の要素の性質 — いくつかの要素に対して真であり、他の要素に対して偽である述語 — を定義すると解釈されるかもしれません。*isEven* 述語は自然数の集合から偶数の部分集合を選びます。トポスでは、述語のアイデアを Ω への対象 a からの射として一般化することができます。これが Ω が時々真実対象と呼ばれる理由です。

述語は論理の構成要素です。トポスには論理を研究するための必要な器具がすべて含まれています。それには論理積(論理かつ)に対応する積、論理和(論理または)に対応する余積、および含意に対応する指數関数が含まれます。論理のすべての標準的な公理は、排中律(または同等に、二重否定の除去)を除いて、トポスで成り立ちます。それが、トポスの論理が構成的または直観主義論理に対応する理由です。

直観主義論理は着実に地位を築いており、コンピュータ科学から予期せぬ支持を得ています。排中律の古典的な概念は、任意の言明が真または偽であるという信念に基づいています。古代ローマ人は *tertium non datur* (第三の選択肢はない) と言いました。しかし、何かが真または偽であるかを知る唯一の方法は、それを証明または反証することです。証明はプロセス、計算です – そして私たちは計算には時間とリソースがかかるのことを知っています。場合によっては、それらは決して終了しないかもしれません。有限時間内にそれを証明できない場合、言明が真であると主張することは意味がありません。より微妙な真実対象を持つトポスは、興味深い論理をモデリングするためのより一般的なフレームワークを提供します。

29.4 チャレンジ

1. 特性関数に沿った *true* の引き戻しである関数 f が単射でなければならぬことを示してください。

30

Lawvere 理論

最 近では、モナドに言及せずに関数型プログラミングについて語ることはできません。しかし、偶然にも、Eugenio Moggi がモナドではなく Lawvere 理論に注目した別の宇宙が存在します。その宇宙を探検しましょう。

30.1 普遍代数

代数を説明する方法は、抽象度の異なる様々な方法があります。私たちは、モノイド、群、環などのような構造を説明する一般的な言語を見つけようとしています。最も単純なレベルでは、これらの構造はすべて、集合の要素に対する操作と、これらの操作によって満たされ

なければならないいくつかの規則を定義します。例えば、結合的な二項演算によってモノイドを定義することができます。また、単位要素と単位則があります。しかし、少し想像力を働かせれば、単位要素を引数を取らない零項演算 – 特定の集合の要素を返す演算に変えることができます。群について話す場合は、要素を取り、その逆要素を返す単項演算子を追加します。それに伴う左右の逆規則があります。環は 2 つの二項演算子といいくつかの追加の規則を定義します。等々です。

大まかなところでは、代数は様々な n の値に対する n 項演算の集合と、恒等式の集合によって定義されます。これらの恒等式はすべて全称量化されています。結合性に関する式は 3 つの要素のすべての可能な組み合わせに対して満たされなければならない、等々です。

偶然にも、これは 0 (加法に関する単位要素) が乗法に関して逆を持たないという単純な理由で、体を考慮外にします。体の可逆則を全称量化することはできません。

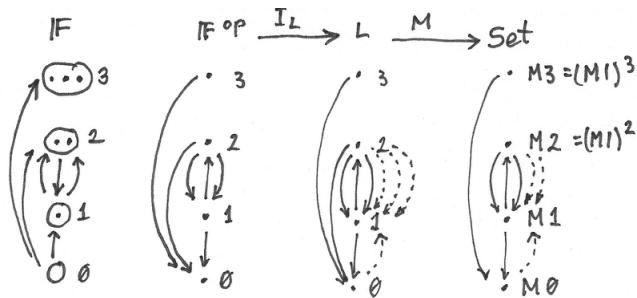
この普遍代数の定義は、操作 (関数) を射に置き換えることにより、**Set** 以外の圏に拡張することができます。集合の代わりに、私たちは一般的な対象 a (一般対象 (generic object) と呼ばれます) を選択します。単項演算は a の自己同型です。しかし、他のアリティ (アリティは与えられた演算の引数の数です) についてはどうでしょうか？ 二項演算 (アリティ 2) は、積 $a \times a$ から a への射として定義することができます。一般的な n 項演算は、 a の n 乗から a への射です：

$$\alpha_n :: a^n \rightarrow a$$

零項演算は終対象 (a の 0 乗) からの射です。したがって、任意の代数を定義するために必要なのは、一つの特別な対象 a のべき乗である対象を持つ圏だけです。特定の代数はこの圏のホム集合にエンコードされています。これが Lawvere 理論の要点です。

Lawvere 理論の導出は多くのステップを経ますので、ここにロードマップを示します：

1. 有限集合の圏 FinSet
2. そのスケルトン \mathbf{F}
3. その逆 \mathbf{F}^{op}
4. Lawvere 理論 L : 圏 Law の対象
5. Lawvere 圏のモデル M : 圏 $\text{Mod}(\text{Law}, \text{Set})$ の対象



30.2 Lawvere 理論

すべての Lawvere 理論は共通の背骨を共有しています。Lawvere 理論のすべての対象は、単に積(実際には、ただのべき乗)を使用してただ一つの対象から生成されます。しかし、一般的な圏でこれらの積をどのように定義しますか？積を定義するために、より単純な圏からのマッピングを使用することができる事が判明しました。実際、このより単純な圏は積の代わりに余積を定義するかもしれません、私たちはそれらを目標の圏に埋め込むために反変関手を使用します。反変関手は余積を積に変え、単射を射影に変えます。

Lawvere 圈の背骨に自然な選択肢は、有限集合の圏、**FinSet** です。これには空集合 \emptyset 、一つの要素を持つ集合 1 、2 つの要素を持つ集合 2 、などが含まれます。この圏のすべての対象は、余積(空集合を特殊な場合の零項余積として扱う)を使用して单一の要素集合から生成することができます。例えば、2 つの要素を持つ集合は 2 つの单一要素の合計で、 $2 = 1 + 1$ として表現されます。これは Haskell で表現されます:

```
type Two = Either () ()
```

```
type Two = Either[Unit, Unit]
```

しかし、空集合がただ一つだと自然に思えるにもかかわらず、多くの異なる单集合が存在するかもしれません。特に、集合 $1 + \emptyset$ は集合

$\emptyset + 1$ とは異なり、 1 とも異なります — それらはすべて同型です。集合の圏における余積は結合的ではありません。この状況を解決するためには、すべての同型集合を識別する圏を構成することができます。このような圏は**スケルトン**と呼ばれます。言い換えれば、任意の Lawvere 理論の背骨は **FinSet** のスケルトン **F** です。この圏の対象は、**FinSet** における要素カウントに対応する自然数（ゼロを含む）として識別することができます。余積は加算の役割を果たします。**F** の射は、有限集合間の関数に対応します。例えば、 \emptyset から n への一意な射があります（空集合が始対象である）、 n から \emptyset への射はありません ($\emptyset \rightarrow \emptyset$ を除く)、 1 から n への n の射があります（単射）、 n から 1 への一つの射、などです。ここで、 n は **F** 内のすべての n 要素集合を **FinSet** で同型を通じて識別される対応する対象です。

F を使用して、*Lawvere 理論*を特別な関手と装備された圏 **L** として形式的に定義することができます:

$$I_{\mathbf{L}} :: \mathbf{F}^{op} \rightarrow \mathbf{L}$$

この関手は対象上での单射でなければならず、有限積を保持しなければなりません (\mathbf{F}^{op} 内の積は **F** 内の余積と同じです) :

$$I_{\mathbf{L}}(m \times n) = I_{\mathbf{L}}m \times I_{\mathbf{L}}n$$

時々、この関手は対象上での恒等関手として特徴づけられることがあります。これは、**F** と **L** の対象が同じであることを意味します。したがって、私たちはそれらに同じ名前を使用します — 自然数で表しま

す。ただし、 \mathbf{F} の対象が集合と同じではないことを念頭に置いてください（それらは同型集合のクラスです）。

\mathbf{L} のホム集合は、一般的に \mathbf{F}^{op} のそれよりも豊かです。それらは、 \mathbf{FinSet} の関数に対応するもの以外の射を含む場合があります（後者は時々**基本積演算**と呼ばれます）。Lawvere 理論の恒等則は、これらの射にエンコードされています。

重要な観察は、 \mathbf{F} 内の単集合 1 が \mathbf{L} 内の何らかの対象 1 にマップされ、 \mathbf{L} 内の他のすべての対象が自動的にこの対象のべき乗になるということです。例えば、 \mathbf{F} 内の 2 つの要素集合 2 は余積 $1 + 1$ であるため、 \mathbf{L} 内で積 1×1 （または 1^2 ）にマップされなければなりません。この意味では、圏 \mathbf{F} は \mathbf{L} の対数のように振る舞います。

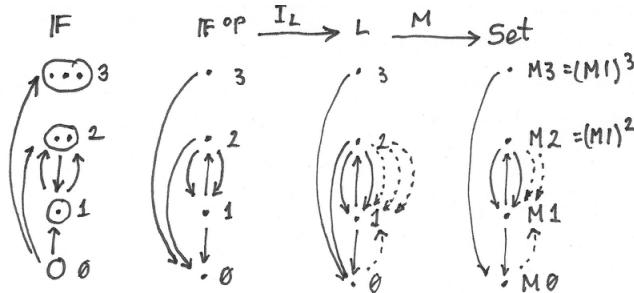
\mathbf{L} の射の中には、関手 $I_{\mathbf{L}}$ によって \mathbf{F} から転送されたものがあります。それらは \mathbf{L} 内で構造的な役割を果たします。特に、余積の単射 i_k は積の射影 p_k になります。射影を想像する便利な直感は、次のように考えることです：

$$p_k :: 1^n \rightarrow 1$$

これは n 変数の関数の原型であり、 k^{th} 変数以外のすべてを無視します。逆に、 \mathbf{F} 内の定数射 $n \rightarrow 1$ は、 \mathbf{L} 内で対角射 $1 \rightarrow 1^n$ になります。これは変数の複製に対応します。

\mathbf{L} 内の興味深い射は、射影以外の n 項演算を定義するものです。これらの射が一つの Lawvere 理論を別のものと区別します。これらは代数を定義する乗算、加算、単位要素の選択などです。しかし、 \mathbf{L} を完全な圏にするためには、 $n \rightarrow m$ （または同等に、 $1^n \rightarrow 1^m$ ）の複合演算も

必要です。圏の単純な構造のために、それらは $n \rightarrow 1$ のタイプのより単純な射の積であることがわかります。これは、積を返す関数が関数の積であるという言明の一般化です（または、以前に見たように、ホム関手は連続です）。



Lawvere 理論 L は F^{op} に基づいており、それから積を定義する「退屈な」射を継承します。それは n 項演算（点線の矢印）を記述する「興味深い」射を追加します。

Lawvere 理論は圏 Law を形成し、その射は有限積を保持し、関手 I と交換する関手です。このような理論が 2 つある場合、 (L, I_L) と $(L', I'_{L'})$ 、それらの間の射は、次のような関手 $F :: L \rightarrow L'$ です：

$$F(m \times n) = Fm \times Fn$$

$$F \circ I_L = I'_{L'}$$

Lawvere 理論間の射は、一つの理論を別の理論内での解釈のアイデアを捉えます。例えば、群の乗法は逆を無視することでモノイドの乗法として解釈することができます。

Lawvere 圈の最も単純な自明な例は、 \mathbf{F}^{op} 自体です ($I_{\mathbf{L}}$ に対する恒等関手の選択に対応します)。この Lawvere 理論には操作や規則がないため、**Law** 内の始対象になります。

この時点では非自明な例を提示するのが非常に役立ちますが、まずモデルが何であるかを理解することなく説明するのは難しいでしょう。

30.3 Lawvere 理論のモデル

Lawvere 理論を理解する鍵は、そのような理論が同じ構造を共有する多くの個々の代数を一般化することを認識することです。例えば、モノイドの Lawvere 理論は、モノイドであることの本質を記述します。それはすべてのモノイドに対して有効でなければなりません。特定のモノイドは、そのような理論のモデルになります。モデルは、Lawvere 理論 \mathbf{L} から集合の圏 \mathbf{Set} への関手として定義されます。(他の圏を使用して Lawvere 理論を一般化することができますが、ここでは \mathbf{Set} に集中しましょう。) \mathbf{L} の構造は積に大きく依存しているため、そのような関手は有限積を保持する必要があります。したがって、 \mathbf{L} のモデル、または Lawvere 理論 \mathbf{L} 上の代数は、次のように関手で定義されます:

$$\begin{aligned} M &:: \mathbf{L} \rightarrow \mathbf{Set} \\ M(a \times b) &\cong Ma \times Mb \end{aligned}$$

積の保存は同型までのということに注意してください。これは非常に重要です。積の厳格な保存は、ほとんどの興味深い理論を排除します。

モデルによる積の保存は、**Set** 内の M のイメージが、集合 $M1$ のべき乗によって生成される一連の集合であることを意味します — 対象 1 からの集合 $M1$ のイメージを呼びましょう。この集合を a とします。(この集合は時々ソートと呼ばれ、そのような代数は**単一ソート**と呼ばれます。Lawvere 理論を多ソート代数に一般化することもあります。) 特に、**L** からの二項演算は関数としてマップされます:

$$a \times a \rightarrow a$$

任意の関手と同様に、**L** 内の複数の射が **Set** 内で同じ関数に折り畳まれる可能性があります。

偶然にも、すべての規則が全称量化された等式であるという事実は、すべての Lawvere 理論が自明なモデルを持つことを意味します: すべての対象を单集合にマップし、すべての射をそれに対する恒等関数にマップする定数関手です。

一般的な射 $m \rightarrow n$ は、関数としてマップされます:

$$a^m \rightarrow a^n$$

もし異なる 2 つのモデル、 M と N があれば、それらの間の自然変換は n によって索引付けられた関数の族です:

$$\mu_n :: Mn \rightarrow Nn$$

または同等に:

$$\mu_n :: a^n \rightarrow b^n$$

ここで $b = N1$ です。

自然性条件は n 項演算の保存を保証します:

$$Nf \circ \mu_n = \mu_1 \circ Mf$$

ここで $f :: n \rightarrow 1$ は \mathbf{L} 内の n 項演算です。

モデルを定義する関手は、モデルの圏 $\mathbf{Mod}(\mathbf{L}, \mathbf{Set})$ を形成し、自然変換を射とします。

自明な Lawvere 圏 \mathbf{F}^{op} のモデルを考えてみましょう。そのようなモデルは、 1 でのその値 $M1$ によって完全に決定されます。 $M1$ が任意の集合である場合、 \mathbf{Set} 内の集合の数だけこれらのモデルがあります。さらに、 $\mathbf{Mod}(\mathbf{F}^{op}, \mathbf{Set})$ 内のすべての射(関手 M と N 間の自然変換)は、 $M1$ でのそのコンポーネントによって一意に決定されます。逆に、 $M1 \rightarrow N1$ の任意の関数は、2 つのモデル M と N の間の自然変換を誘導します。したがって $\mathbf{Mod}(\mathbf{F}^{op}, \mathbf{Set})$ は \mathbf{Set} と同値です。

30.4 モノイドの理論

最も単純な非自明な例としてモノイドの Lawvere 理論があります。これは、この理論のモデルがモノイドの全圏 \mathbf{Mon} をスパンするすべての可能なモノイドの構造を蒸留する单一の理論です。私たちはすでに普遍構成を見てきました。これは、任意のモノイドが適切な自由モノイドから一部の射を識別することによって得られることを示しました。したがって、单一の自由モノイドはすでに多くのモノイドを一般

化しています。しかしながら、無限に多くの自由モノイドがあります。モノイドの Lawvere 理論 L_{Mon} は、それらを一つのエレガントな構造に組み合わせます。

すべてのモノイドには単位が必要なので、 L_{Mon} 内には 0 から 1 への特別な射 η がなければなりません。 F 内に対応する射は存在しないことに注意してください。そのような射は反対の方向、つまり FinSet 内で单集合から空集合への関数になります。そのような関数は存在しません。

次に、 $2 \rightarrow 1$ への射を考えます。これらは $L_{\text{Mon}}(2, 1)$ のメンバーであり、すべての二項演算の原型を含まなければなりません。 $\text{Mod}(L_{\text{Mon}}, \text{Set})$ 内でモデルを構成するとき、これらの射はデカルト積 $M1 \times M1$ から $M1$ への関数にマップされます。言い換えれば、2つの引数を持つ関数です。

問題は、モノイド演算子を使用して実装できる 2 引数の関数がどれだけあるかです。2 つの引数を a と b としましょう。両方の引数を無視してモノイド単位を返す関数が 1 つあります。次に、それぞれ a と b を返す 2 つの射影があります。それに続いて、 ab 、 ba 、 aa 、 bb 、 aab などを返す関数があります… 実際には、 a と b のジェネレーターを持つ自由モノイドの要素と同じ数の 2 引数関数があります。 $L_{\text{Mon}}(2, 1)$ は、そのようなすべての射を含まなければなりません。なぜなら、そのモデルの 1 つは自由モノイドだからです。自由モノイドでは、それらは異なる関数に対応します。他のモデルは、 $L_{\text{Mon}}(2, 1)$ 内の複数の

射を单一の関数に折り畳むかもしれません、自由モノイドではそうではありません。

n ジェネレーターを持つ自由モノイドを n^* と表記すると、ホム集合 $L(2, 1)$ をモノイドの圏 \mathbf{Mon} 内のホム集合 $\mathbf{Mon}(1^*, 2^*)$ と同一視することができます。一般に、 $L_{\mathbf{Mon}}(m, n)$ を $\mathbf{Mon}(n^*, m^*)$ とします。言い換えれば、圏 $L_{\mathbf{Mon}}$ は自由モノイドの圏の逆です。

モノイドの Lawvere 理論のモデルの圏、

$\mathbf{Mod}(L_{\mathbf{Mon}}, \mathbf{Set})$ は、すべてのモノイドの圏、 \mathbf{Mon} と同値です。

30.5 Lawvere 理論とモナド

覚えているかもしれません、代数理論はモナドを使用して記述することができます – 特にモナドに関する代数。それでは、Lawvere 理論とモナドの間には接続があることに驚くことはありません。

まず、Lawvere 理論がモナドをどのように誘導するかを見てみましょう。それは、忘却関手と自由関手の間の随伴関係を通じて行われます。忘却関手 U は、各モデルに集合を割り当てます。この集合は、 $\mathbf{Mod}(L, \mathbf{Set})$ からの関手 M を L の対象 1 で評価することによって与えられます。

U を導出する別の方法は、 F^{op} が \mathbf{Law} 内の始対象であるという事実を利用することです。つまり、任意の Lawvere 理論 L に対して、一意の関手 $F^{op} \rightarrow L$ があります。この関手は、モデルに対する逆関手を誘

導します (モデルは理論から集合への関手です) :

$$\mathbf{Mod}(\mathbf{L}, \mathbf{Set}) \rightarrow \mathbf{Mod}(\mathbf{F}^{op}, \mathbf{Set})$$

しかし、私たちが議論したように、 \mathbf{F}^{op} のモデルの圏は \mathbf{Set} と同値なので、忘却関手を取得します:

$$U :: \mathbf{Mod}(\mathbf{L}, \mathbf{Set}) \rightarrow \mathbf{Set}$$

このように定義された U は常に左随伴、自由関手 F を持つことが示されます。

これは有限集合に対して容易に見ることができます。自由関手 F は自由代数を生成します。自由代数は、 $\mathbf{Mod}(\mathbf{L}, \mathbf{Set})$ 内の特定のモデルであり、生成器の有限集合 n から生成されます。 F は表現可能関手として実装することができます:

$$\mathbf{L}(n, -) :: \mathbf{L} \rightarrow \mathbf{Set}$$

それが実際に自由であることを示すためには、それが忘却関手の左随伴であることを証明するだけです:

$$\mathbf{Mod}(\mathbf{L}(n, -), M) \cong \mathbf{Set}(n, U(M))$$

右側を単純化しましょう:

$$\mathbf{Set}(n, U(M)) \cong \mathbf{Set}(n, M1) \cong (M1)^n \cong Mn$$

(集合の射の集合が指数と同型であるという事実を使用しました。この場合、それは単に反復された積です。) 随伴は米田の補題の結果です:

$$[\mathbf{L}, \mathbf{Set}](\mathbf{L}(n, -), M) \cong Mn$$

一緒に、忘却と自由関手は **Set** 上のモナド $T = U \circ F$ を定義します。したがって、すべての Lawvere 理論はモナドを生成します。

実は、このモナドの代数の圏は、モデルの圏に同値です。

モナドに関する代数は、モナドを使用して形成された式を評価する方法を定義します。Lawvere 理論は n 項演算を使用して式を生成する方法を定義します。モデルは、これらの式を評価する手段を提供します。

ただし、モナドと Lawvere 理論の接続は双方向ではありません。限定的なモナドのみが Lawvere 理論につながります。限定的なモナドは限定的な関手に基づいています。限定的な関手は **Set** 上でその有限集合への作用によって完全に決定されます。任意の集合 a へのその作用は、次の余を使用して評価できます:

$$Fa = \int^n a^n \times (Fn)$$

余は余積、または合計を一般化しますので、この式は幕級数展開の一般化です。または、関手が一般化されたコンテナであるという直感を使用することができます。その場合、 a の限定的なコンテナは、形と内容の合計として記述できます。ここで、 Fn は n 要素を格納するための形の集合であり、内容は a^n の要素の n 組自体です。たとえば、リスト(関手として)は限定的であり、各アリティに 1 つの形を持ちます。ツリーにはアリティごとにより多くの形があります、などです。

まず、Lawvere 理論から生成されるすべてのモナドは限定的であり、余エンドとして表現できます:

$$T_L a = \int^n a^n \times L(n, 1)$$

逆に、**Set** 上の任意の限定的なモナド T が与えられた場合、Lawvere 理論を構成することができます。 T の Kleisli 圏を構成することから始めます。思い出してください、Kleisli 圏の射は、基礎となる圏内の射によって与えられます:

$$a \rightarrow Tb$$

有限集合に制限された場合、これは次になります:

$$m \rightarrow Tn$$

この Kleisli 圏の逆圏、 Kl_T^{op} を有限集合に制限すると、問題の Lawvere 理論になります。特に、 L 内で n 項演算を記述するホム集合 $L(n, 1)$ は、 $Kl_T(1, n)$ のホム集合によって与えられます。

実際、プログラミングで遭遇するほとんどのモナドは限定的ですが、継続モナドが注目に値する例外です。限定的な操作を超えて Lawvere 理論の概念を拡張することは可能です。

30.6 余エンドとしてのモナド

余エンドの式をもう少し詳しく探ってみましょう。

$$T_L a = \int^n a^n \times L(n, 1)$$

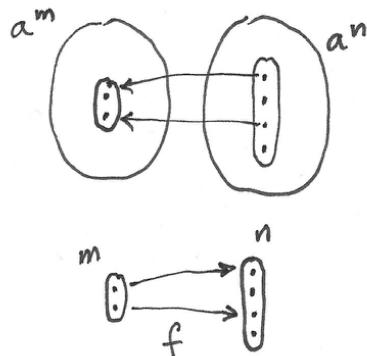
始めに、この余エンドは \mathbf{F} 内のプロ関手 P に対して取られます:

$$Pnm = a^n \times \mathbf{L}(m, 1)$$

このプロ関手は、最初の引数 n で反変です。それが射をどのように持ち上げるかを考えましょう。 \mathbf{FinSet} 内の射は有限集合のマッピング $f :: m \rightarrow n$ です。このマッピングは n 要素集合から m 要素を選択することを説明します(繰り返しが許可されます)。それは a のべき乗へのマッピングに持ち上げられます、つまり(方向に注意してください):

$$a^n \rightarrow a^m$$

持ち上げは単純に n 組の要素から m 要素を選択します(繰り返しを伴って) (a_1, a_2, \dots, a_n) 。



たとえば、 $f_k :: 1 \rightarrow n$ を取ります。これは n 要素集合から k^{th} 要素を選択することを説明します。それは、 a の n 組から k^{th} 要素を返す関数に持ち上げられます。

または、 $f :: m \rightarrow 1$ を取ります。これはすべての m 要素を一つにマッピングする定数関数です。その持ち上げは、 a の单一要素を取り、それを m 回複製する関数です:

$$\lambda x \rightarrow (\underbrace{x, x, \dots, x}_m)$$

このプロ関手が第二引数において共変であることは直ちに明らかではありません。ホム関手 $\mathbf{L}(m, 1)$ は実際には m において反変です。しかしながら、余エンドは \mathbf{L} の圏ではなく \mathbf{F} の圏で取られます。余エンド変数 n は有限集合（またはそのようなもののスケルトン）を渡ります。圏 \mathbf{L} には \mathbf{F} の逆が含まれているので、 \mathbf{F} の射 $m \rightarrow n$ は \mathbf{L} の $\mathbf{L}(n, m)$ のメンバーです（埋め込みは関手 $I_{\mathbf{L}}$ によって与えられます）。

\mathbf{F} から \mathbf{Set} への関手として $\mathbf{L}(m, 1)$ の関手性を確認しましょう。関数 $f :: m \rightarrow n$ を持ち上げることを目指していますので、 $\mathbf{L}(m, 1)$ から $\mathbf{L}(n, 1)$ への関数を実装することが目標です。関数 f に対応する \mathbf{L} 内の射は n から m へです（方向に注意してください）。この射を $\mathbf{L}(m, 1)$ と事前合成することで、 $\mathbf{L}(n, 1)$ の部分集合を得ます。

$$\mathbf{L}(m, 1) \longrightarrow \mathbf{L}(n, 1)$$

$${}^{m_{\bullet}} \xrightarrow[f]{} {}^{\bullet^n}$$

$1 \rightarrow n$ の関数を持ち上げることにより、 $\mathbf{L}(1, 1)$ から $\mathbf{L}(n, 1)$ へ行くことができます。後でこの事実を使用します。

プロ関手 a^n と共に変関手 $\mathbf{L}(m, 1)$ の積は、 $\mathbf{F}^{op} \times \mathbf{F} \rightarrow \mathbf{Set}$ へのプロ関手です。余エンドは、プロ関手のすべての対角コンポーネントの非交和 (disjoint sum) として定義され、いくつかの要素が特定されます。これらの識別は、余くさび条件に対応します。

ここでは、余エンドはすべての n にわたる集合 $a^n \times \mathbf{L}(n, 1)$ の非交和として始まります。識別は、プロ関手の非対角項 $a^n \times \mathbf{L}(m, 1)$ から出発して、対角線に到達するために、積の最初のコンポーネントまたは二番目のコンポーネントに射 $f :: m \rightarrow n$ を適用することによって生成されます。2つの結果はその後識別されます。

$$\begin{array}{ccc}
 & a^n \times \mathbf{L}(m, 1) & \\
 \langle f, \text{id} \rangle \swarrow & \sim & \searrow \langle \text{id}, f \rangle \\
 a^m \times \mathbf{L}(m, 1) & & a^n \times \mathbf{L}(n, 1)
 \end{array}$$

$$f :: m \rightarrow n$$

以前に示したように、 $f :: 1 \rightarrow n$ を持ち上げることにより、次の2つの変換が得られます:

$$a^n \rightarrow a$$

そして:

$$\mathbf{L}(1, 1) \rightarrow \mathbf{L}(n, 1)$$

したがって、 $a^n \times \mathbf{L}(1, 1)$ から始めて、次の両方に到達できます：

$$a \times \mathbf{L}(1, 1)$$

$\langle f, \mathbf{id} \rangle$ を持ち上げるときと、

$$a^n \times \mathbf{L}(n, 1)$$

$\langle \mathbf{id}, f \rangle$ を持ち上げるときです。これは、 $a^n \times \mathbf{L}(n, 1)$ のすべての要素が $a \times \mathbf{L}(1, 1)$ と識別されるわけではないことを意味します。 $\mathbf{L}(n, 1)$ のすべての要素が $\mathbf{L}(1, 1)$ から基本射を適用することで到達できるわけではありませんからです。 \mathbf{F} からの射を持ち上げることによってのみ構成できる非自明な n 項演算は \mathbf{L} にあります。

言い換えると、余エンドの式に含まれるすべての加数を識別することができます。これは、 $\mathbf{L}(n, 1)$ が基本射を適用することによって $\mathbf{L}(1, 1)$ から到達できる場合に限ります。それらはすべて $a \times \mathbf{L}(1, 1)$ に等価です。基本射は、 \mathbf{F} の射のイメージです。

\mathbf{F}^{op} 自体の Lawvere 理論の最も単純な場合を見てみましょう。この理論では、すべての $\mathbf{L}(n, 1)$ が $\mathbf{L}(1, 1)$ から到達できます。これは $\mathbf{L}(1, 1)$ がただの恒等射だけを含む一つの集合であり、 $\mathbf{L}(n, 1)$ が \mathbf{F} 内の单射 $1 \rightarrow n$ に対応する射だけを含むからです。これらは**基本射**です。したがって、余エンドのすべての加数は同等であり、結果は次のようになります：

$$Ta = a \times \mathbf{L}(1, 1) = a$$

これは恒等モナドです。

30.7 副作用の Lawvere 理論

モナドと Lawvere 理論の間に強い関連があるので、Lawvere 理論がモナドの代わりにプログラミングで使用されるかどうかという疑問が自然に浮かびます。モナドの主な問題点は、それらがうまく構成されないことです。モナドトランسفォーマーを構成するための一般的なレシピがありません。この分野では Lawvere 理論が優位性を持ちます：それらは余積やテンソル積を使用して構成することができます。一方、限定的なモナドのみが容易に Lawvere 理論に変換できます。ここでの例外は継続モナドです。この分野では現在も研究が進行中です。

副作用を記述するために Lawvere 理論をどのように使用できるかの例を示すために、通常は **Maybe** モナドを使用して実装される例外の単純なケースを議論します。

Maybe モナドは、单一の零項演算 $0 \rightarrow 1$ を持つ Lawvere 理論によって生成されます。この理論のモデルは、 1 をいくつかの集合 a にマップし、零項演算を関数にマップする関手です：

```
raise :: () -> a
```

```
def raise: Unit => A
```

余エンドの式を使用して **Maybe** モナドを回復することができます。零項演算の追加がホム集合 $L(n, 1)$ にどのように影響するかを考えてみましょう。新しい $L(0, 1)$ を作成するだけでなく (F^{op} には存在しな

い)、 $\mathbf{L}(n, 1)$ に新しい射を追加します。これらは、タイプ $n \rightarrow 0$ の射と $0 \rightarrow 1$ の私たちの射を合成することによって得られます。このような寄与は、余エンドの式の中で $a^0 \times \mathbf{L}(0, 1)$ と識別されます。それらは、次から取得できます:

$$a^n \times \mathbf{L}(0, 1)$$

$0 \rightarrow n$ を 2 つの異なる方法で持ち上げることにより。

$$\begin{array}{ccc} & a^n \times \mathbf{L}(0, 1) & \\ \langle f, \mathbf{id} \rangle \swarrow & \sim & \searrow \langle \mathbf{id}, f \rangle \\ a^0 \times \mathbf{L}(0, 1) & & a^n \times \mathbf{L}(n, 1) \end{array}$$

$$f :: 0 \rightarrow n$$

余エンドは次のように簡約化されます:

$$T_{\mathbf{L}} a = a^0 + a^1$$

または、Haskell の表記法を使用して:

```
type Maybe a = Either () a  
  
type Option[A] = Either[Unit, A]
```

これは次と同等です:

```
data Maybe a = Nothing | Just a

sealed trait Option[+A]
case object None extends Option[Nothing]
case class Some[A](a: A) extends Option[A]
```

この Lawvere 理論は例外の発生のみをサポートしており、その処理はサポートしていません。

30.8 チャレンジ

1. **F** (**FinSet** のスケルトン) 内で 2 と 3 の間のすべての射を列挙してください。
2. モノイドの Lawvere 理論のモデルの圏が、リストモナドのモナド代数の圏に同値であることを示してください。
3. モノイドの Lawvere 理論はリストモナドを生成します。対応する Kleisli 射を使用してその二項演算を生成できることを示してください。
4. **FinSet** は **Set** の部分圏であり、それを **Set** に埋め込む関手があります。**Set** 上の任意の関手は **FinSet** に制限することができます。限定的な関手がその制限の左 Kan 拡張であることを示してください。

30.9 参考文献

1. Functorial Semantics of Algebraic Theories^{*1}, F. William Lawvere
2. Notions of computation determine monads^{*2}, Gordon Plotkin and John Power

^{*1} <http://www.tac.mta.ca/tac/reprints/articles/5/tr5.pdf>

^{*2} http://homepages.inf.ed.ac.uk/gdp/publications/Comp_Eff_Monads.pdf

31

モナド、モノイド、そして圏

卷

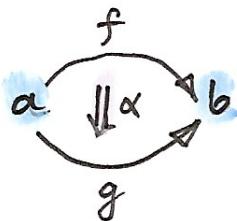
論に関する本を終わらせる最適な場所はありません。学ぶべきことは常にあります。圏論は広大な主題です。同時に、同じテーマ、概念、パターンが何度も何度も現れることが明らかです。すべての概念が Kan 拡張であるという言葉があります。実際、Kan 拡張を使用して極限、余極限、随伴、モナド、米田の補題などを導き出すことができます。圏の概念自体は、あらゆる抽象化レベルで現れ、モノイドやモナドの概念も同様です。どれが最も基本的かというと、実はすべてが相互関連しており、一つが次へつながる終わりのない抽象化のサイクルになっています。これらの相互接続を示すことが、この本を終わらせる良い方法かもしれません。

31.1 双圈

圏論の最も難しい側面の一つは、視点の絶え間ない切り替えです。たとえば、集合の圏を考えてみましょう。私たちは、要素の観点から集合を定義することに慣れています。空集合には要素がありません。単集合には要素が一つあります。二つの集合の直積は、ペアの集合です、等々。しかし、集合の圏 **Set** について話すとき、集合の内容を忘れて、その間の射(矢印)に集中するように求めました。たまには、特定の普遍構成が **Set** の要素の観点から何を記述しているのかを覗き見ることが許されていました。終対象は一つの要素を持つ集合であることがわかります、等々。しかし、これらは単なる健全性チェックでした。

関手は圏の写像として定義されます。写像を圏の射として考えるのは自然なことです。関手は圏の圏(小さな圏であれば、サイズに関する質問を避けることができます)の射としてわかりました。関手を射として扱うことにより、圏の内部(その対象と射)に対するその作用の情報を放棄することになります。これは、集合の要素に対する関数の作用の情報を放棄するのと同じです、それを **Set** の射として扱うときです。しかし、任意の二つの圏間の関手もまた圏を形成します。今度は、一つの圏の射だったものを、別の圏の対象として考えるように求められます。関手圏では、関手は対象であり、自然変換は射です。同じものが、一つの圏では射であり、別の圏では対象であることがわかりました。対象を名詞、射を動詞とする素朴な視点は成り立ちません。

二つの視点を切り替える代わりに、一つに統合しようとすることができます。こうして 2-圏の概念が得られます。ここでは、対象を 0-セル、射を 1-セル、射間の射を 2-セルと呼びます。



0-セル a, b ; 1-セル f, g ; そして 2-セル α .

圏の圏 **Cat** はすぐに思いつく例です。ここでは、圏が 0-セル、関手が 1-セル、自然変換が 2-セルです。2-圏の規則は、任意の二つの 0-セル間の 1-セルが圏を形成することを教えてくれます（言い換えると、 $C(a, b)$ はホム集合ではなくホム圏です）。これは、任意の二つの圏間の関手が関手圏を形成するという以前の主張とよく合います。

特に、任意の 0-セルからそれ自身への 1-セルも圏、ホム圏 $C(a, a)$ を形成します。しかし、この圏はさらに多くの構造を持っています。 $C(a, a)$ のメンバーは、C の射として、または $C(a, a)$ の対象として見ることができます。射として、それらは互いに合成することができます。しかし、対象として見たとき、合成は対象のペアから対象への写像となります。実際、それは積一正確にはテンソル積一と非常に似ています。このテンソル積には単位があります：恒等 1-セルです。実際

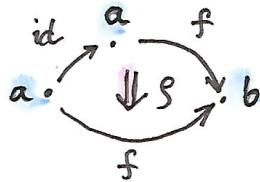
には、任意の 2-圏のホム圏 $C(a, a)$ は、1-セルの合成として定義されたテンソル積を持つモノイダル圏と自動的になります。結合則と単位則は、対応する圏の規則から単純に導かれます。

さて、このことが我々の標準的な 2-圏 Cat の例で何を意味するのかを見てみましょう。ホム圏 $\text{Cat}(a, a)$ は、 a の上の自己関手の圏です。自己関手の合成は、テンソル積の役割を果たします。恒等関手は、この積に対する単位です。以前、自己関手がモノイダル圏を形成することを見ました（これはモナドの定義に使用されました）、しかし今、これがより一般的な現象であることがわかります：任意の 2-圏の自己 1-セルはモノイダル圏を形成します。後で、モナドを一般化するときに、これに戻ってきます。

一般的なモノイダル圏では、モノイド則が厳密に満たされる必要はないことを覚えているかもしれません。単位則と結合則が同型まで満たされれば十分でした。2-圏では、 $C(a, a)$ のモノイダル則は 1-セルの合成則から導かれます。これらの規則は厳密なので、常に厳密なモノイダル圏を得ます。ただし、これらの規則を緩和することも可能です。たとえば、恒等 1-セル \mathbf{id}_a と別の 1-セル、 $f :: a \rightarrow b$ の合成が、 f と等しいという代わりに、 f と同型であると言うことができます。1-セルの同型は 2-セルを使用して定義されます。言い換えると、以下のような 2-セルがあります。

$$\rho :: f \circ \mathbf{id}_a \rightarrow f$$

それは逆を持っています。



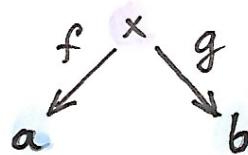
双圏の恒等性は同型(可逆な2-セル ρ)まで成立します。

左恒等性と結合則についても同じことができます。このような緩和された2-圏は双圏と呼ばれます(ここでは省略しますが、いくつかの追加の整合性規則があります)。

予想通り、双圏の自己1-セルは非厳密な規則を持つ一般的なモノイダル圏を形成します。

双圏の興味深い例は、スパンの圏です。二つの対象 a と b の間のスパンは、対象 x と一対の射のペアです:

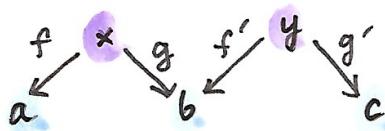
$$\begin{aligned} f &:: x \rightarrow a \\ g &:: x \rightarrow b \end{aligned}$$



圏論的な積の定義にスパンを使用したことを思い出すかもしれません。ここでは、双圏の1-セルとしてスパンを見てみたいと思います。

スパンの合成を定義する最初のステップはです。隣接するスパンを持つていると仮定します:

$$\begin{aligned}f' &:: y \rightarrow b \\g' &:: y \rightarrow c\end{aligned}$$



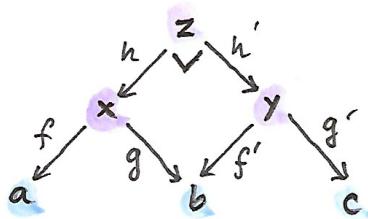
合成は第三のスパンであり、その頂点は何らかの z です。それに対する最も自然な選択は、 g と f' の引き戻しです。引き戻しとは、 z と二つの射のペアです:

$$\begin{aligned}h &:: z \rightarrow x \\h' &:: z \rightarrow y\end{aligned}$$

そのような:

$$g \circ h = f' \circ h'$$

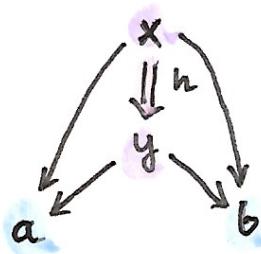
これは、すべてのそのような対象の中で普遍的なものです。



今のところ、集合の圏におけるスパンに集中しましょう。この場合、引き戻しは単にペア (p, q) の集合です $x \times y$ からで、次のようにになります:

$$g\ p = f'\ q$$

二つのスパンが同じ終点を共有している場合のスパン間の射は、それらの頂点間の射 h として定義され、適切な三角形が可換であることが求められます。



スパン内の 2-セル Span.

要約すると、双圏 **Span** では: 0-セルは集合、1-セルはスパン、2-セルはスパン射です。恒等 1-セルは、すべての三つの対象が同じで、二つの射が恒等である退化したスパンです。

以前にも双圏の例を見ました: プロ関手の双圏 **Prof** では、0-セルは圏、1-セルはプロ関手、2-セルは自然変換です。プロ関手の合成は余エンドによって与えられました。

31.2 モナド

この時点で、モナドが自己関手の圏のモノイドとしての定義にかなり慣れているはずです。双圏 **Cat** の小さいホム圏の自己 1-セルのみならず、モナドの新しい理解を持ってこの定義を再訪しましょう。自己関手の圏はモノイダル圏です: テンソル積は自己関手の合成から来ます。モノイドは、ここでは終関手 T を持つモノイダル圏の対象として定義されます—二つの射と共に。射は自己関手間の射です、すなわち自然変換です。一つの射はモノイダル単位—恒等自己関手—を T に写します:

$$\eta :: I \rightarrow T$$

二つ目の射は $T \otimes T$ のテンソル積を T に写します。テンソル積は自己関手の合成によって与えられるので、私たちは得ます:

$$\mu :: T \circ T \rightarrow T$$

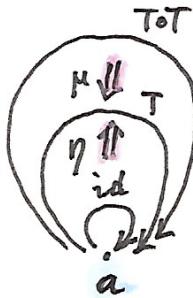
$$\begin{array}{ccc}
 & T \circ T & \\
 & \downarrow \mu & \\
 I & \xrightarrow{\eta} & T
 \end{array}$$

これらはモナドを定義する二つの操作として認識します (Haskell では `return` と `join` と呼ばれます)、そしてモノイド則はモナド則になります。

さて、この定義から自己関手の言及をすべて取り除きましょう。双圈 C を始めにして、それにおける 0-セル a を選びます。以前に見たように、ホム圏 $C(a, a)$ はモノイダル圏です。したがって、 $C(a, a)$ におけるモノイドを定義することができます。1-セル、 T を選び、そして二つの 2-セルを選びます:

$$\begin{aligned}
 \eta &: I \rightarrow T \\
 \mu &: T \circ T \rightarrow T
 \end{aligned}$$

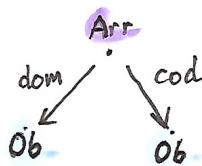
モノイド則を満たすようにします。これをモナドと呼びます。



これは、0-セル、1-セル、そして2-セルを使ってのモナドのはるかに一般的な定義です。それは双圏 \mathbf{Cat} に適用されるとき通常のモナドに簡約されます。しかし、他の双圏で何が起こるかを見てみましょう。

双圏 \mathbf{Span} におけるモナドを構成しましょう。0-セルを選びます。これは、すぐに明らかになる理由から、私が Ob と呼ぶことにする集合です。次に、自己 1-セルを選びます: Ob から Ob へのスパンです。頂点にある集合を Ar と呼びます。二つの関数で装備されています:

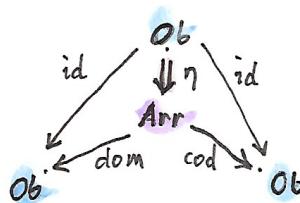
$$\begin{aligned} dom &:: Ar \rightarrow Ob \\ cod &:: Ar \rightarrow Ob \end{aligned}$$



集合 Ar の要素を「射」と呼びましょう。また、 Ob の要素を「対象」と呼ぶことにはすれば、どこに行くのかヒントが得られるかもしれません。二つの関数 dom と cod は、「射」に始域と終域を割り当てます。

私たちのスパンをモナドにするためには、二つの 2-セル、 η と μ が必要です。この場合のモノイダル単位は、頂点が Ob にある Ob から Ob への自明なスパンです、そして二つの恒等関数です。2-セル η は、頂点 Ob と Ar の間の関数です。言い換えると、 η は「対象」ごとに「射」を割り当てます。 Span 内の 2-セルは、可換性条件を満たさなければなりません—この場合は：

$$\begin{aligned} dom \circ \eta &= \mathbf{id} \\ cod \circ \eta &= \mathbf{id} \end{aligned}$$



コンポーネントで表すと、これは以下のようになります：

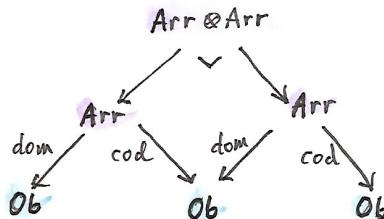
$$dom(\eta ob) = ob = cod(\eta ob)$$

ここで、 ob は Ob の「対象」です。言い換えると、 η は始域と終域がその「対象」である「射」をすべての「対象」に割り当てます。この特別な「射」を「恒等射」と呼びましょう。

二つ目の 2-セル μ は、スパン Ar の自己合成に作用します。合成は引き戻しとして定義されるので、その要素は Ar の要素のペア－「射」のペア (a_1, a_2) です。引き戻し条件は次のようにになります:

$$cod\ a_1 = dom\ a_2$$

私たちは a_1 と a_2 が「合成可能」であると言います。なぜなら、一方の始域が他方の終域です。



2-セル μ は、合成可能な射のペア (a_1, a_2) を集合 Ar の単一の射 a_3 に写します。言い換えると μ は射の合成を定義します。

モナド則が射の恒等性と結合性の規則に対応することを確認するの簡単です。私たちはちょうど圈(小さな圈です、対象と射が集合を形成します)を定義しました。

つまり、圈はただの双圈のスパンにおけるモナドです。

この結果の驚くべき点は、それが圈をモナドやモノイドなどの他の代数的構造と同じ立場に置くことです。圈であることには何も特別なことはありません。それは単に二つの集合と四つの関数です。実際、対象のために別の集合が必要とされるわけではなく、対象は恒等射

(一対一の対応関係にあります) と同一視することができます。ですから、それは実際にはただの集合といいくつかの関数です。数学のあらゆる分野で圏論が果たす中心的な役割を考えると、これは非常に謙虚な実現です。

31.3 チャレンジ

1. 双圏における自己 1-セルの合成として定義されるテンソル積の単位則と結合則を導出してください。
2. **Span** 内のモナドのモナド則が、結果の圏における恒等性と結合性の規則に対応することを確認してください。
3. **Prof** 内のモナドが対象上同一の関手であることを示してください。
4. **Span** 内のモナドのモナド代数とは何か？

31.4 参考文献

1. Paweł Sobociński のブログ^{*1}。

^{*1} <https://graphicallinearalgebra.net/2017/04/16/a-monoid-is-a-category-a-category-is-a-monad-a-monad-is-a-monoid/>

索引

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer.

—Donald E. Knuth, *Fundamental Algorithms*

(Volume 1 of *The Art of Computer Programming*)

curry 化, 193

Lawvere 理論, 594

modus ponens, 206, 207

poset, 69

rig, 115

writer モナド, 63

くさび条件, 524

による一意性, 68

アドホック多相性, 217

アリティ, 591

インスタンス, 134

コンポーネント, 212

スケルトン, 594

テンソル積, 439

テンプレートテンプレートパラメータ,
137

デカルト閉圏, 199

トポス, 581

バリエント, 86

パラメトリック多相性, 174, 217

プレモノイダル, 156

プロ関手, 179

ボトム, 20

モノイダル圏, 98

レコード, 101

- 一对一, 91
一方向, 346
上への, 91
不動点, 481
余イコライザ, 534
全单射, 92, 325
全射, 91, 325
全関数, 91
内部, 187
初期代数, 484
半環, 114
单一ソート, 598
单射, 91, 325
双デカルト閉圈, 200
双関手, 154
反変, 226
台となる, 285
台集合, 366
右随伴, 362
同値, 346
同値関係, 536
同型, 344, 373
因数分解器, 83
型推論, 17
埋め込み, 91
基本積演算, 595
対象, 331
対象 (object), 2
射 (morphism), 3
左随伴, 346
忘却関手, 286, 366
恒等射 (identity), 7
持ち上げられた, 335
指数関数, 197
操作的意味論 (operational semantics), 22
文脈的計算, 452
普遍錐, 255
水平合成, 236
準同型, 284, 483
無料の定理, 174
環, 114
矢印 (arrow), 2
等価性, 343
等式推論, 127
米田埋め込み, 326
純粹関数 (pure function), 26
自己関手, 350
自然, 340
自然に同型, 346
自然同型, 215
自然性条件, 213
自由モノイド, 281
自由関手, 366
表現, 297
表現可能, 297
表現可能前層, 262
表示的意味論 (denotational semantics), 23
証明関連関係, 519
評価, 187
豊穣, 292
述語 (predicate), 32
逆圏, 72
関数適用, 187

謝辞

数学と論理のチェックをしてくれた Edward Kmett 氏と Gershom Bazerman 氏に感謝します。また、多くのボランティアが私の間違いを訂正し、本を改善する手助けをしてくれたことにも感謝しています。

私の C++ のモノイドコンセプトコードを、彼と Bjarne Stroustrup 氏の最新の提案に基づいて書き直してくれた Andrew Sutton 氏に感謝します。

ドラフトを読んでくれて、C++14 の高度な機能を使って型推論を進める **compose** の賢い実装を提供してくれた Eric Niebler 氏に感謝します。これによって、型特性を使って同じことをしていた古風なテンプレートマジックの全セクションを削除できました。さようなら、古い魔法たち！

奥付

この本は、Bartosz Milewski のオリジナルテキストを LATEX 形式に変換することによって、Igal Tabachnik^{*2}氏によって編纂されました。まずは Mercury Web Parser^{*3}を使ってオリジナルの WordPress ブログ投稿をスクレイピングし、クリーンな HTML コンテンツを取得、Beautiful Soup^{*4}を使用して修正・調整し、最終的に Pandoc^{*5}で LATEX に変換しました。

本文の書体は、Philipp H. Poll 氏による Linux Libertine で、見出しへには Linux Biolinum を使用しています。タイプライター書体は、Raph Levien 氏によって作成され、Dimosthenis Kaponis 氏と Takashi Tanigawa 氏によって Inconsolata LGC の形で補完されています。表

^{*2} <https://hmemcpy.com>

^{*3} <https://mercury.postlight.com/web-parser/>

^{*4} <https://www.crummy.com/software/BeautifulSoup/>

^{*5} <https://pandoc.org/>

紙の書体は、Juan Pablo del Peral 氏によってデザインされた Alegreya です。

オリジナルの本のレイアウトデザインとタイポグラフィは Andres Raba 氏によって行われました。シンタックスハイライトは、Hugo Maia Vieira^{*6}氏による「GitHub」スタイルの Pygments を使用しています。Scala コードの翻訳は Typelevel^{*7}の貢献者によって行われました。

^{*6} <https://github.com/hugomaiavieira/pygments-style-github>

^{*7} https://github.com/typelevel/CT_from_Programmers.scala

コピーレフトに関する通知

この本は自由なライセンスに従っており、フリーソフトウェア^{*8}の哲学に基づいています: この本を好きなように利用することができます。ソースは公開されています。また、この本を再配布したり、あなた自身のバージョンを配布することもできます。つまり、印刷したり、コピーしたり、メールで送ったり、ウェブサイトにアップロードしたり、変更したり、翻訳したり、リミックスしたり、一部を削除したり、その上に何かを描いたりすることができます。

この本はコピーレフトです: 本を変更して自分のバージョンを配布する場合、受け取る人たちにもこれらの自由を認めなければなりません。この本は Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0) を使用しています。

^{*8} <https://www.gnu.org/philosophy/free-sw.en.html>

