計算機科学のための圏論の基礎の基礎

Akihiko Koga

Ver. 0.90, 2020.3.25

私のホームページ
http://www.cs-study.com/koga/
に関連の情報がありますので
ご興味があればそちらも参照してみてください.
また, この資料のメンテナスもそこで行う予定です.

内容

- 動機
- 圏論の概要（集合論との比較）
- 圏論の基礎概念の導入
- 型付きλ計算
- 型付きλ計算とCCC の関係
- さらなる学習に向けて
- 参考文献

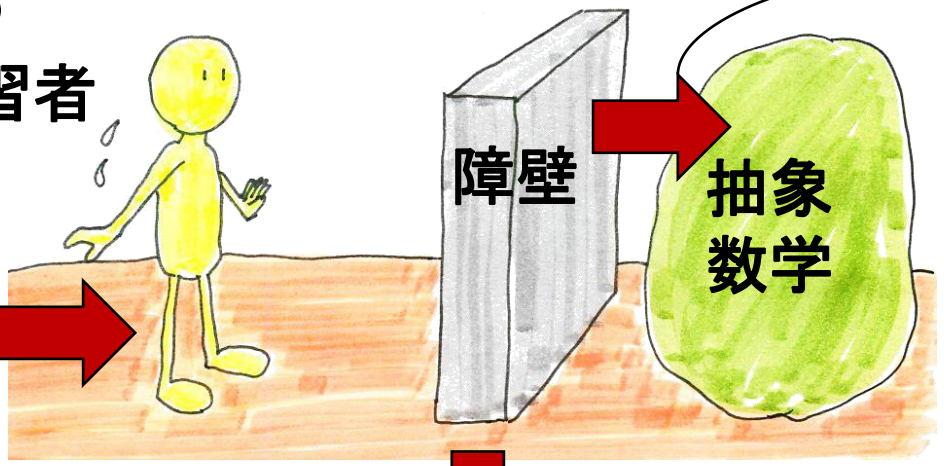
動機

- 圏論の計算機応用に関するありそうな感想
  - 1970年頃から**圏論が計算機科学でも**使われ始め，発表も**年々増加**しているらしい
  - なんとなく高度なことができそうだが．．．
    - 成果も圏論の言葉で語られるので本当に**使える成果が出ているか判断できない**
    - 身近な成果として**「モナド」**があるらしいが，**圏論とどう関係するのか分からない**
  - 圏論が分かるようになりたいが，習得にはかなりの**数学の習熟度が要りそう**
    中々**手がでない**．あるいは，すでに**挫折した．それも何度か**

- この資料を作成した動機
  - 今後，**圏論が**計算機科学の必須の基礎になるかどうかは確信はないが，**廃れていくことはない**と思う
  - 本当に計算機科学で役に立つかどうか見極めるには，**圏論が分かる人口が今の数倍に増える必要**があるのではないだろうか？
  - しかし，通常の学習者が持っている離散数学くらいの知識では**挫折しやすい**
  - それらの学習者が，**市販の圏論の教科書を読み始められる**ようなところまで持っていく**ブースター的な教材**があると良いのではないだろうか

本資料の作成方針

- **応用指向の学習者**が**抽象数学の学習**を成功させるためには
  次の3つが重要と考える



①学習者を引きつける

その学習の先に豊富な応用があることを匂わせる

学習したいという動機付け

③足腰を鍛える

抽象的な概念を使って推論したり，証明したりできる体力をつける

②障壁を引き下げる

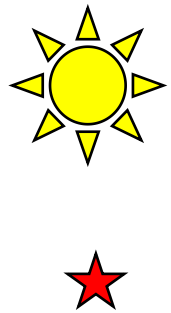
抽象的な概念を，図解，意図の解説，例の提示，比喩などを使ってなじみやすくする

（http://www.cs-study.com/koga/lattice/index.html の絵を再利用した）

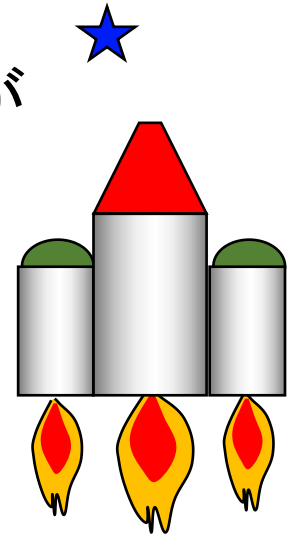
本資料の作成方針

- 逆に言えば，現状は前頁の3つの条件が欠けていると思う

  - 圏論が何に利用できるのか分からない
  - 登場する概念の意味や意図が分からない
  - 圏論の中の抽象的な概念の構成方法に頭がついていけない

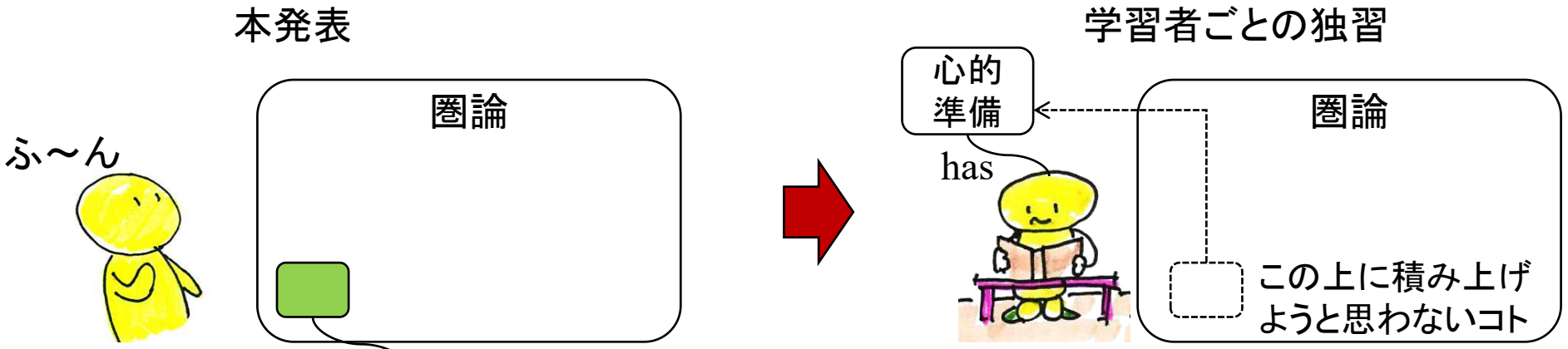


- これらを解決した教材を整備して，興味のある学習者が自ら学べるところまで行ける**ブースター**を提供できたら良いと思う



本資料の作成方針

- 圏論の中の非常に小さい部分で良いので，**応用の香りが入っている部分を選び**，その中に**登場する概念の徹底的な理解容易化**を図る
- 今回は2時間と短いので**「足腰の鍛錬」**は**割愛**する（でも，**重要**）
- 目標としては，その上に圏論の学習を積み上げるのではなく，一般の教科書で最初から学習を始めるための**心的な準備**を行うこととする



応用の香りが入っているごく**狭い部分**を選び，登場する**概念の徹底的な理解容易化**を図る

動機や**核となる概念**のいくつかに**馴染みを持った状態**で，自分で頑張って勉強する

注意事項

- 後の各自の学習では，学習者自身による**足場の組直しが必要**
  - 今回の圏論の諸概念の理解容易化では，私自身の特殊な心的イメージを使っている．したがって，かなり**バイアスの掛かった説明**になっている可能性がある
  - **概念の把握方法は人に寄る**ので学習者によってはふさわしくないかもしれない
  - もともと学習は**各自がそれぞれのイメージを確立していく**過程であると考える
  - したがって，今回の説明は**各自が学習に取り掛かるまでの仮の足場**にはなっても，それを保持し続けるのは**弊害が出てくる可能性**がある

- 今回はかなり**範囲を絞った**ので，この後，**補わないといけない**基礎的な概念が**かなり**ある
  - 特に**「自然変換」**を外した．これはもともと，圏論が考え出される動機となった概念であり，また，**圏論を高階に進めていく上ではキー**となる概念である
  - 自分で学習するときは，この概念の把握がキーとなる

本日の発表の内容

- 今回の資料の主な内容

  Cartesian Closed Category (CCC) と呼ばれる圏が**単純型付きλ計算**のモデルになるということを説明する

- 補足
  - **単純型付きλ計算**は次の2つの操作が可能で，かつ，式に**型**をつけることができる簡単なプログラミング言語である
    - 式の中の入力を明示し，関数にする（**抽象化，パラメータの明示化**）
    - 関数を式に適用する（**関数適用**）

例：$(\lambda x{:}int.(x+3))(4)$

整数 x を取って x+3 を返す関数

4 に適用

  - このような式を解釈し，**「意味（semantics）」**を与えることができる圏を構成する
  - そのような圏は**関数適用**と**抽象化**を解釈できる圏であり，圏論の用語でいうと**積 A × B** と**冪 A → B** を持つ圏である（それぞれが正確に対応する訳ではない）
  - まず，圏についての大まかな説明をし，**なんとなくのイメージ**を持てるようにする
  - 次に，積と冪が定義できるだけの**必要最小限の圏の概念**を導入する
  - 続いて，**単純型付きλ計算を導入**し，**CCC がそのモデルになる**ことを示す

内容(再掲)

- 動機
- 圏論の概要(集合論との比較)
- 圏論の基礎概念の導入
- 型付きλ計算
- 型付きλ計算とCCC の関係
- さらなる学習に向けて
- 参考文献

圏論の概要
（集合論との比較）

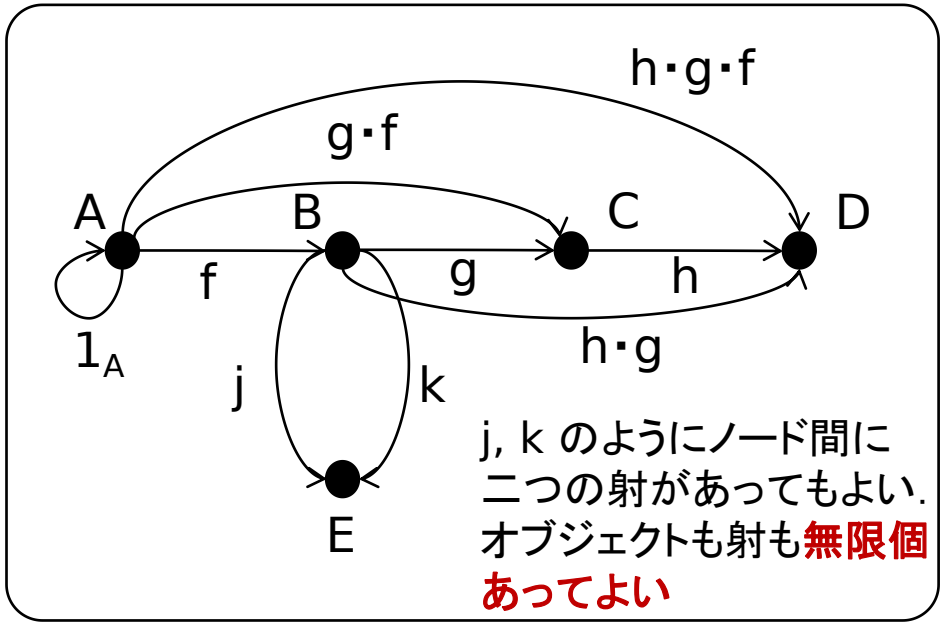
圏論の概要
集合論との比較によるなんとなくの圏論のイメージ把握

- **定義　圏（Category）**

> **圏**とは，**隣あったエッジの合成**が定義されている**マルチエッジ有向グラフ**のこと

- ノードを**オブジェクト（object）**，有向エッジを**射（arrow）**という
- ただし，各オブジェクト X には**恒等射（identity）**と呼ばれる射 $\mathbf{1_X}$ があり，また，射の合成規則は**結合則**を満たす

圏の例



- A, B, C, D, E が**オブジェクト**（ノード）
- f, g, h, … が**射**（有向エッジ）
- 各オブジェクトには**恒等射**がある
（オブジェクトA には $\mathbf{1_A}$，図では他の
オブジェクトに対しては省略した）
- 隣り合った射 f と g の合成を**g･f** と書く
- 射の合成は**結合律を満たす**
(h･g)･f = h･(g･f)
- 1_X は**射の合成について単位元**になる
f･$\mathbf{1_A}$ = f　　$\mathbf{1_B}$･f=f

集合論と圏論の対比

- 学習者が最初に知りたいのは，**横着な疑問**ではあるが

  「圏論とは何か？」

  だと思う.

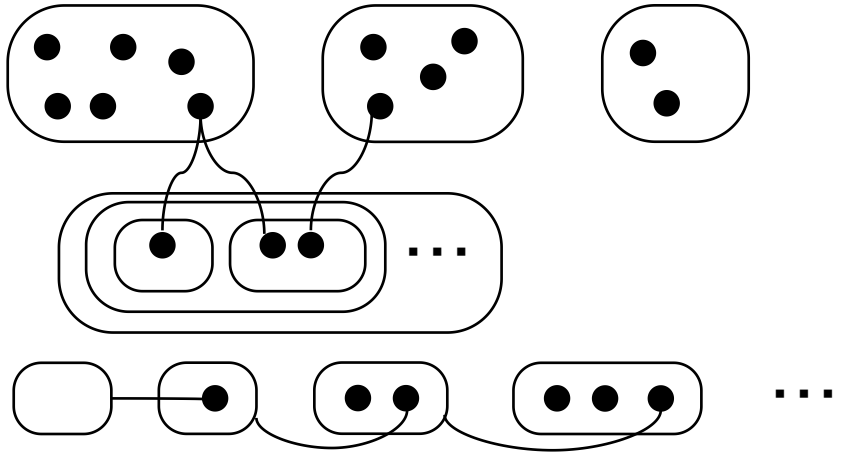
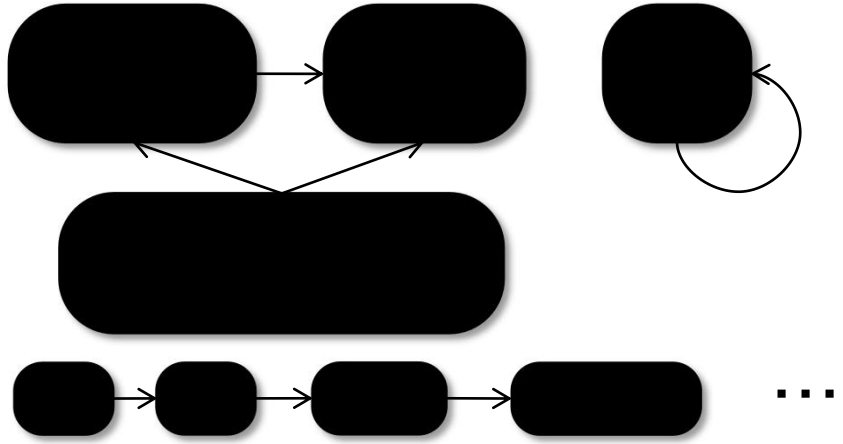
- 圏論は，群論など他の**多用途の数学的な概念**と同様に色々な見方ができるので，ある**確定的な見方は間違いに陥りやすい**

- しかし，導入が難しいのも確かなので，後で壊すつもりで，**一時的なビュー**を作ってみることも良いと思う

- これも色々問題はあるが，**集合論との典型的な対比**を見てみる

  （集合論を集合の圏 **Set** と見たときの対比になっている）

集合論と圏論の対比

	集合論	圏論
理論の様子		
説明	・モノ（**集合**）の内部に**粒々**（**要素**）が属している（∈）関係で数学の概念を次々に構成 ■自然数，整数，有理数，実数，複素数 ■関係，関数，無限集合の理論．．．	・モノの内部は基本的には見えない． ・モノの間（モノの外）の関係（**射**）で数学的概念を規定する．
長所	・**∈関係**を使った**単純な原理だけ**で数学の全体を**具体的に**構成できる	・数学的概念の**（要求）仕様**が書ける （詳細にとらわれず横の関係が見えやすい）
短所	・**実装しすぎる**きらいがある （横の関係が見えなくなることがある）	・具体的なものを書こうとすると，途端に**記述が複雑かつ多量に**なってしまう

集合論と圏論の対比
集合は「実装しすぎるきらいがある」ことの説明

- **「関数」**の概念の例
  - **「関数」**の**素朴なアイデア**はモノの集まりとモノの集まりの**要素同士の対応**
  - 集合論では
    - **順序対 (a, b)** を定義
      例えば**クラトウスキーの順序対** (a, b) := **{{a}, {a, b}}**
    - 集合の**直積 A×B** := {(a, b) | a∈A, b∈B} を定義
    - A と B の**関係**を **R⊆A×B** と定義
    - 関係の中で**一意性を持った関係**を**関数**と定義
  - つまり**関数の実体**は**クラトウスキーの順序対のある集合**
  - しかし，**順序対の定義方法は一通りではない**．上の定義は，その中の1つの方法で関数を**実装してみただけ**
  - 集合で数学的概念を定義するということはこのように**特殊な実装方法を1つ示すことになる**場合が多い
  - 人間が数学をするときは，その特殊な**実装方法は無意識に無視**され，もとの素朴なアイデアのもとに推論を進めている
  - もっと，人間の**素朴なアイデアに近い言葉で数学をしても良いのでは？**

集合論と圏論の対比
実装と仕様記述

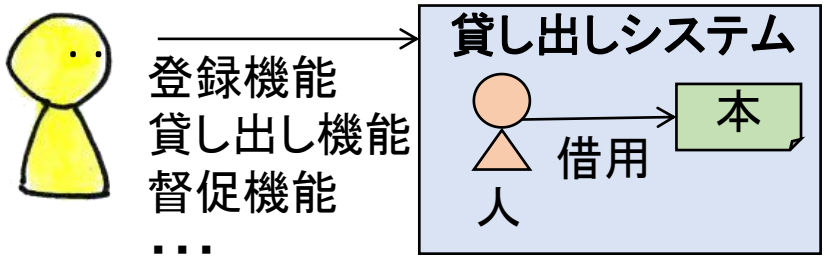
ある種の人たちの要望

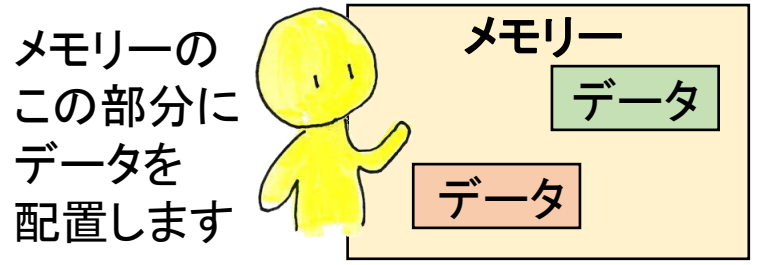
- **集合論**は数学的概念の**実装(implementation)の記述言語**．見通し良く数学を記述するためにはそのもとにある(人による)**要求仕様(specification)を記述する言語**が**必要**ではないか？

- M. Barr & C. Wells “Category Theory for Computing Science”, 1998 では明確に，implementation と specification という言葉を使っている

- c.f.**「図書館の貸し出しシステム」**を開発している状況を考えてみる

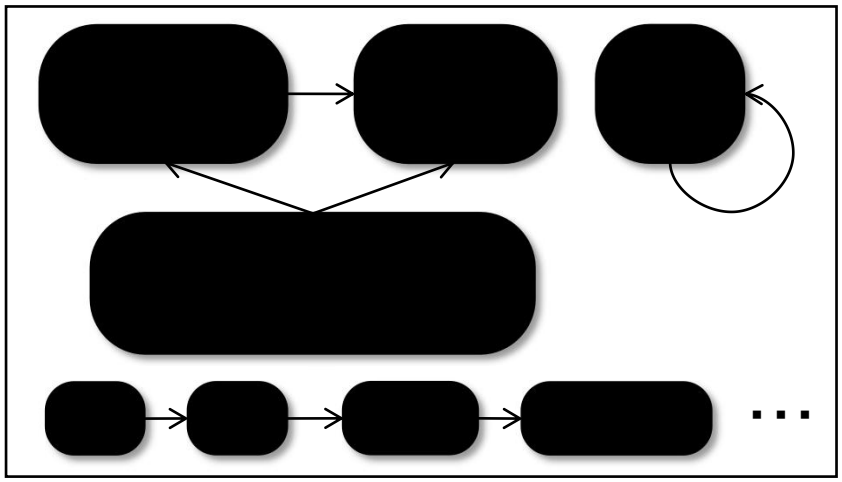
  多くの関係者間の合意を得るための記述は，**どんな Entity があってそれにどのような操作が行われるか等**であり，「本」の**データをメモリー上にどのように配置するかではない**





集合論と圏論の対比

- **圏論**では，集合論で見えていた**容器の中**の粒々を**塗りつぶし**て，外側の**モノ（オブジェクト）の間の関係（射）だけ**で，対象システムを記述する
- **モノの理解の仕方**は次のように変化する



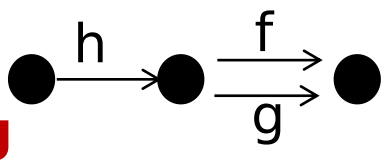
それが**どの要素からできているか**
（集合論の外延性の原理）

⇩

その圏の中のモノから**どんな射を受けて，どんな射が出ていくか**
それらの**射はどんな性質**を持つか

メソッドの集合で**クラスを規定**する
オブジェクト指向に少し似ている
（ただし，圏論では**内部状態は無い**）

- 例：

上への関数（集合論）	epi（圏論）（上への関数に相当）
h : A → B が**上への**関数 ∀b∈B ∃a∈A h(a)=b （**h(a) で B が埋め尽くされている**）	h : A → B が**epi** ∀f, g **h·f=h·g ならば f=g** 

集合論と圏論の対比

- **圏論**が**集合論**の**「実装しすぎ」**に対する**正しい回答かどうか**は分からないが，結果として，内部のゴテゴテしたものが見えにくくなり，全体の**見通しが良くなる**場合がある

- 個人的な意見だが，**詳細な記述が必要な場合**は，粒々がなく，記述力が弱い分だけ，**膨大な記述が必要**になってくるように思う

  - これは **Java** などで，**抽象クラス**や**インタフェース**を**多用**した**フレームワーク**の**記述が膨大**になり，**素人にはとっつきにくくなる**のと似ている．もっとも，フレームワークを作るとき，つい種々の汎用化を考えてしまうのも大きな理由と思うが．

  - 例えば，**圏論では**，集合論と同じような粒々を意識した**部分構造を記述**するためには，**subobject classifier** などの**道具立て**をすることが**必要**となり，そこに至るための基礎知識として本一冊を必要とする（**Topos** のことを意識）

  - **圏論の定義や定理**は**条件が多かったり，複雑だった**りすることが多く，
    これが**初心者の学習を妨げている**一因だと思う

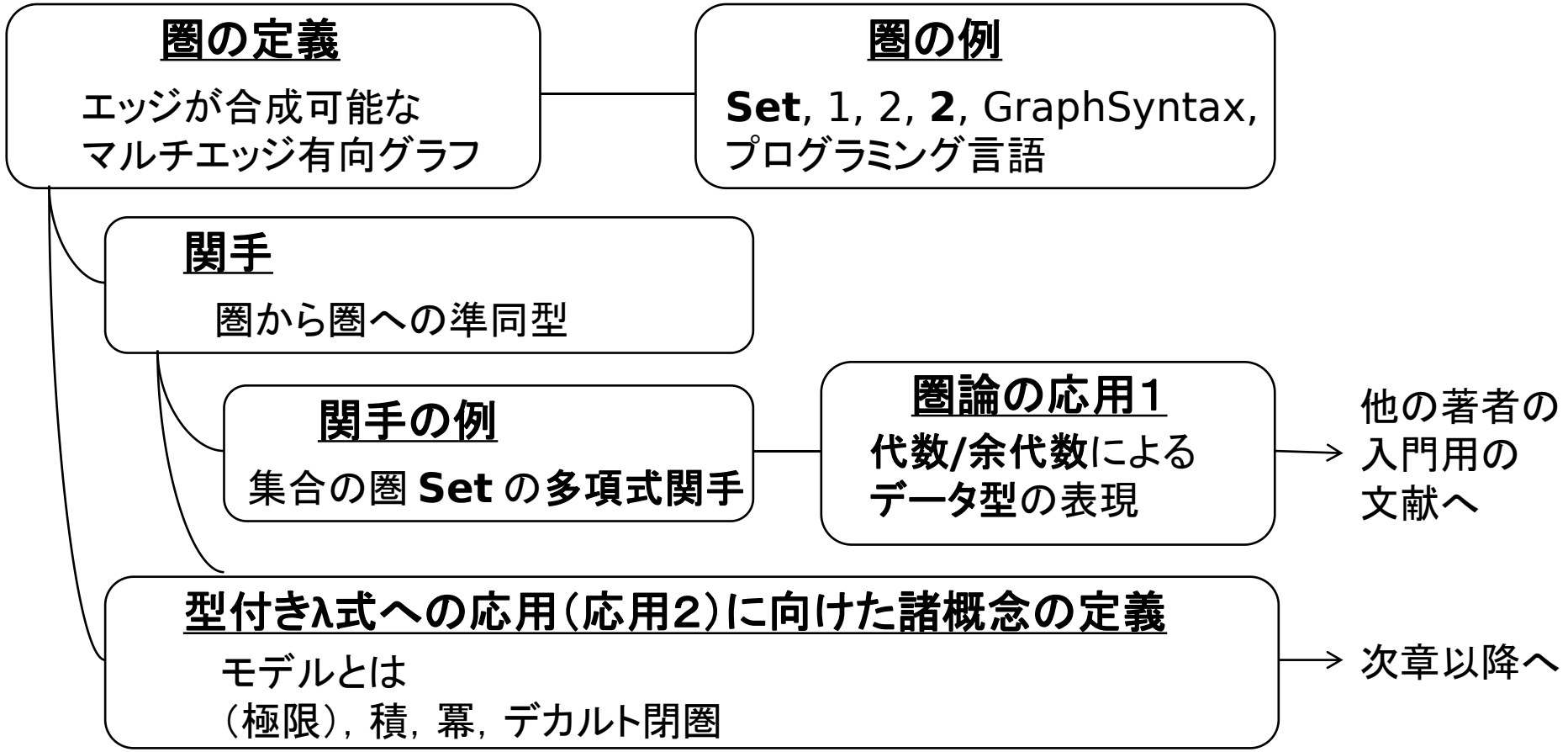
⇒ **この解決が学習の困難さ克服のカギ**

概念を出来るだけ
分かりやすくする工夫と
各自の足腰の鍛錬

圏論の
基礎的概念の導入

ここでのトピックス

ここでは，単純型付き λ計算のモデルとしてのデカルト閉圏を定義するための最小限度の概念を説明する．また，関手の概念を使ってデータ型を表す応用があることも示す．



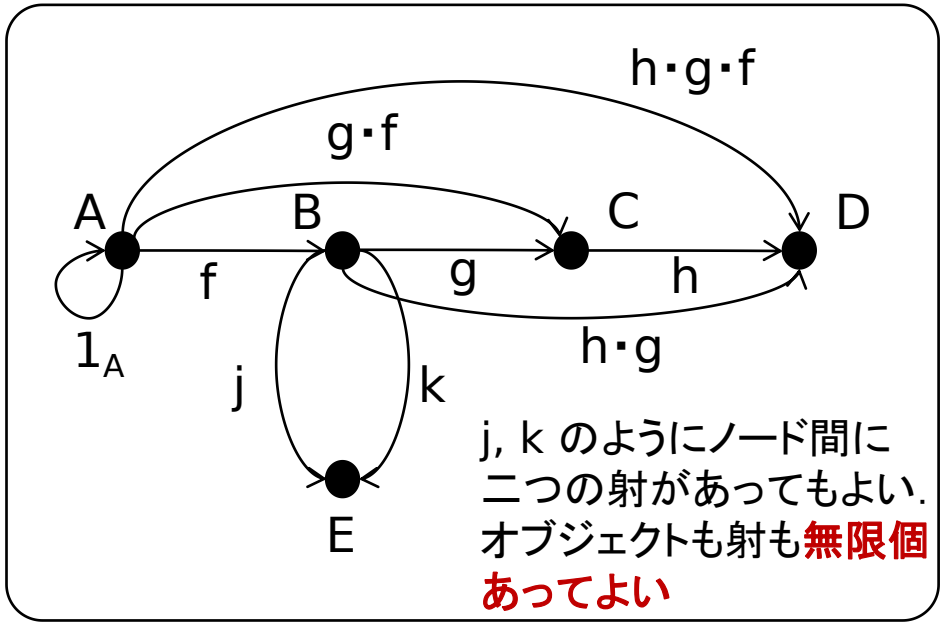
圏論の概要（再掲）

- **定義　圏（Category）**

圏とは，**隣あったエッジの合成**が定義されている **マルチエッジ 有向グラフ**のこと

- ノードを**オブジェクト（object）**，有向エッジを**射（arrow）**という
- ただし，各オブジェクト X には**恒等射（identity）**と呼ばれる射 $\mathbf{1_X}$ があり，また，射の合成規則は**結合則**を満たす

圏の例



･A, B, C, D, E が**オブジェクト**（ノード）
･f, g, h, … が**射**（有向エッジ）

･各オブジェクトには**恒等射**がある
（オブジェクトA には $\mathbf{1_A}$，図では他の
オブジェクトに対しては省略した）

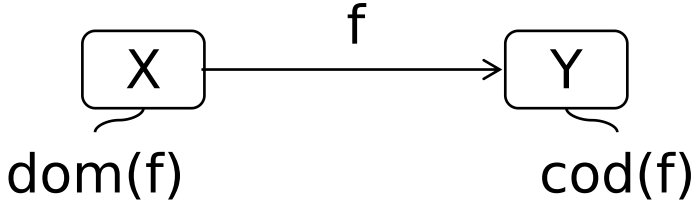
･隣り合った射 f と g の合成を**g･f** と書く

･射の合成は**結合律を満たす**
(h･g)･f = h･(g･f)

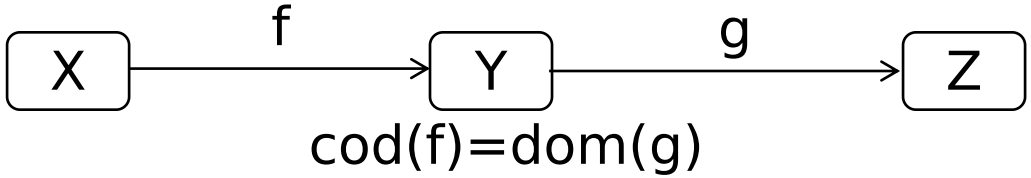
･1_X は**射の合成について単位元**になる
$\mathbf{f･1_A = f}$　$\mathbf{1_B･f=f}$

圏の定義：用語の補足

- **圏のオブジェクトと射についての記法**
  - **C** を圏とするとき，**C** は**オブジェクトの集まり**と，**射の集まり**からなるが，それぞれを $\mathbf{C}_0$，あるいはobj(**C**)，$\mathbf{C}_1$ あるいは arr(**C**) で表す．
  - X が圏 **C** のオブジェクトである時，$X \in \mathbf{C}_0$ あるいは単に $X \in \mathbf{C}$ と書く
  - f が圏 **C** の射である時，$f \in C_1$ あるいは単に $f \in \mathbf{C}$ と書く
- **domain** と **codomain** について
  - 射 $f:X \to Y$ は $X \xrightarrow{f} Y$ とも書く
  - このときXをfのdomain，Yをfのcodomain といい，dom(f), cod(f) と書く



  - 射 f と g が合成できるためには cod(f)=dom(g) であることが必要十分条件



圏の例

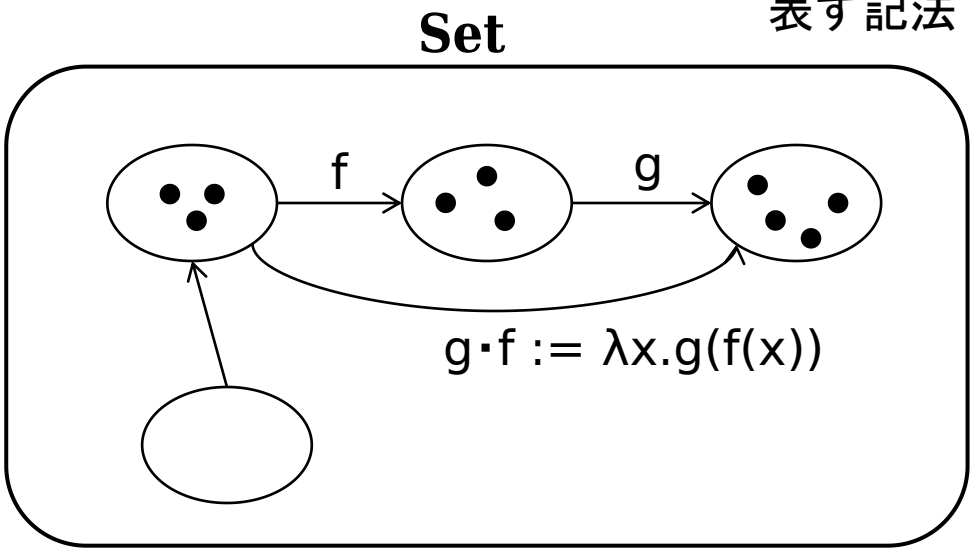
- いくつか圏の例を見ていく

1. 集合の圏 **Set**
2. 集合の圏の一部分の圏
3. 前順序集合（preorder），部分順序集合（poset：partially ordered set）
4. 小さな図式いくつか（1, 2, **2**, GraphSyntax）
5. 関数型言語の圏

圏の例：集合の圏 Set

- **集合の圏 Set**
  - オブジェクト
    - 集合
  - 射
    - 集合から集合への関数
    - 射の合成は通常の関数の合成

      g･f := λx.g(f(x))　　注意）λx.exp(x) は，x を入力としてexp(x)を返す関数を表す記法



Set すべての集合の集まりなので，クラスになるが，今日のところはあまりここに神経質になる必要はない

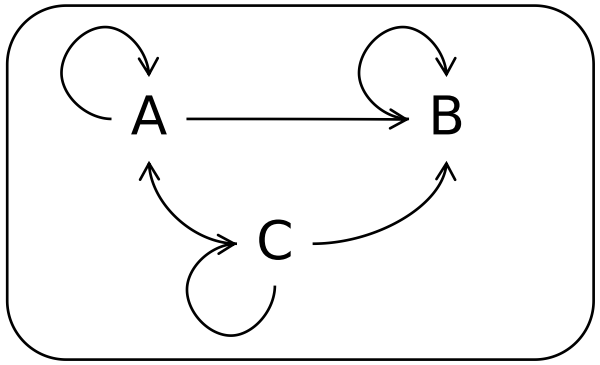
本によっては，**Sets** と書く流儀もある（S. Awodey など）．

圏の例：集合の圏の一部分の圏

- 圏**C**があるとき，その**オブジェクトの一部分**を集め，集めたオブジェクト間の**すべての射**を集めると，それはまた圏になる

- したがって次のようなものは **Set** の部分の圏である
  - **FinSet** … 有限集合とそれらの間の関数の圏

  - A, B, C を集合とするとき，A, B, C をオブジェクトとし，それらの間の関数を射とした圏



オブジェクト：A, B, C
射： A, B, Cの間の関数すべて
（図では書ききれないのでほとんど省略）

射に関しては，それらの**合成が全部含まれるように絞れば**，元の圏の部分的な圏ができる

圏の例：前順序集合，部分順序集合

- **定義** 前順序集合と部分順序集合

  - 集合 P と，その上の2項関係 ≤ の対 (P, ≤) が次の2つの条件を満たす時，(P, ≤) を**前順序集合**（**preordered set**）という．
    - [反射律]　x≤x for ∀x∈P
    - [推移律]　x≤y かつ y≤z ならば x≤z for ∀x, y, z∈P

  - さらに (P, ≤) が上の2つの条件に加えて，次の条件を満たす時，(P, ≤) を**部分順序集合**（partially ordered set あるいは **poset**）という
    - [反対称律] x≤y かつ y≤x ならば　x=y for ∀x, y∈P

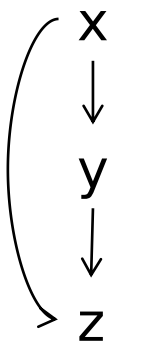
「反対称律」という用語は「≤」から「＝」を除いて「＜」による関係で考えると分かりやすいだろう

[反対称律]　x ＜ y ならば y ＜ x は成り立たない

圏の例：前順序集合，部分順序集合

- 前順序集合および部分順序集合は次のようにして圏とみることができる
  - (P, ≤) を**前順序集合**とする
    - **オブジェクト**
      - P の要素
    - **射**
      - オブジェクト x, y が x≤y のとき，かつ，のときに限り他と区別のつく射 x→y が1つあるとする．
        前順序集合は，推移律が成り立つので，射の連鎖は合成され，他と区別のつく1つの射になる

(P, ≤)

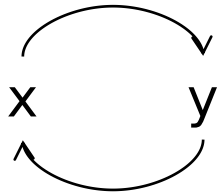


・x≤y のときxからyに1つの射があると考える．

・そうでないときは射はない．

・前順序集合は推移律を満たすので，連鎖した射の合成ができてそれらのオブジェクト間の唯一の射になる

  - **部分順序集合** (P, ≤)も同様に圏とみることができる
    - 次の点だけが違う
      部分順序集合ではお互いに相手に向かって射のある異なるオブジェクトはないこと



なら x≤y かつ y≤x なので x=y

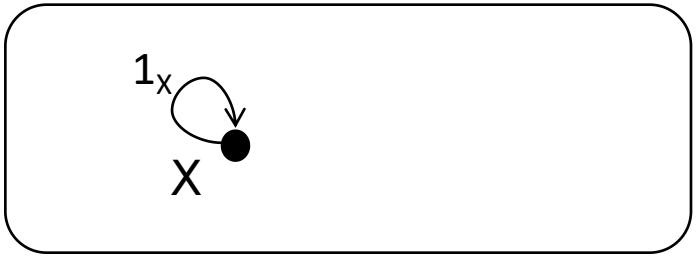
圏の例：前順序集合，部分順序集合
補足

- ここで述べたのは，**前順序集合**や**部分順序集合そのものを圏として見る方法**であり，このような集合の集まりの圏（**Preorder** や **Poset**）ではない
- 前順序集合や部分順序集合そのものの圏は，無理やり例として出したように感じるかもしれないが，とても**重要な圏**である．
- 今日は述べないが，**モノイドを圏としてみる見方**と共に，**圏論の二つの極端な性質**を表現する役割を担っている

- 実際，前順序集合を圏としてみたものは，数学の中で「**ガロア接続**」という概念と結びつき，一つの**研究領域を形成**している
- 圏の新しい概念を吟味するとき，**まず前順序集合や部分順序集合で吟味してみるのがコツ**であるらしい

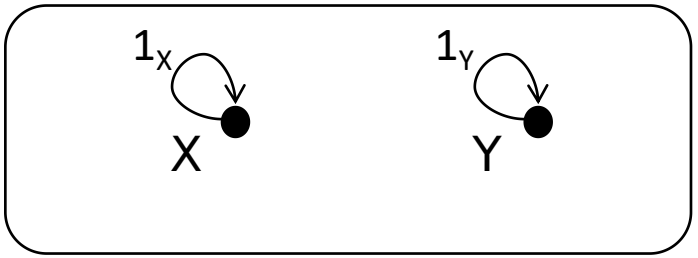
圏の例：小さな図式いくつか（1）

- 1



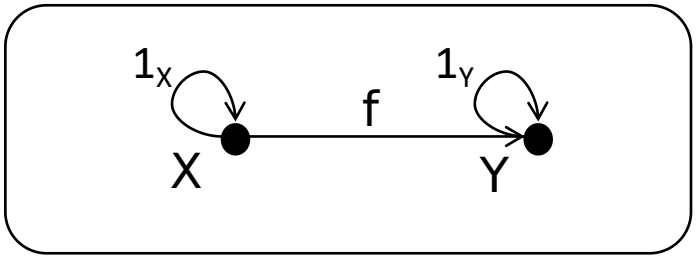
1個のオブジェクトとその恒等射だけからなる圏

- 2



2個のオブジェクトとそれらの恒等射だけからなる圏．二つのオブジェクトの間にはなんの射もない．

- **2**

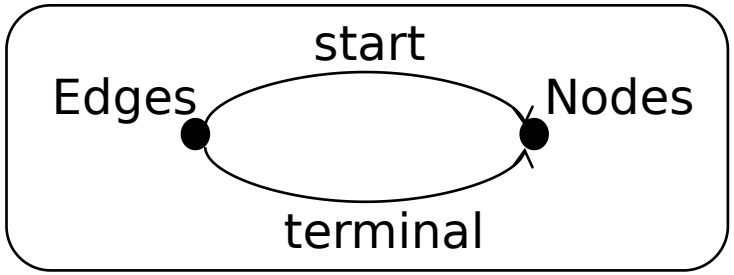


2個のオブジェクトとそれらの恒等射，片方のオブジェクトから他方への1つの射からなる圏．

以下，図において恒等写像は省略する

圏の例：有限の圏いくつか（2）

- **GraphSyntax**

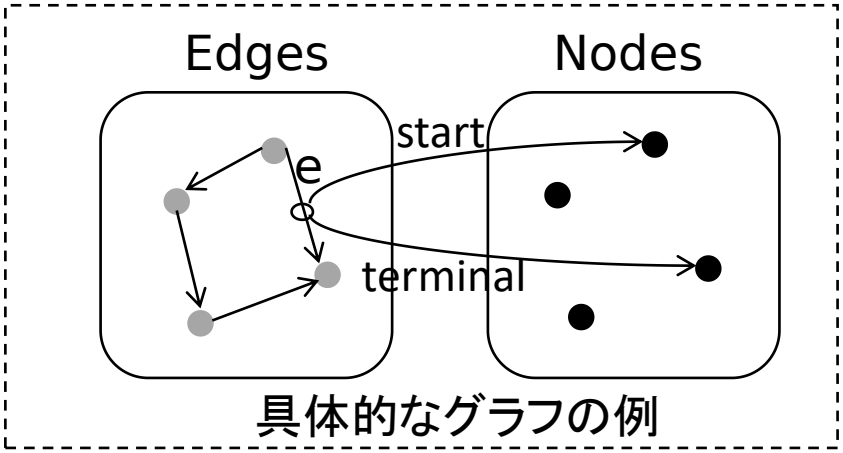


前頁の **2** に射が1本加わった形の圏

- グラフというものの**骨格（文法）**を表そうとしている．
  つまり，グラフとは
  - エッジの集合 Edges と
  - ノードの集合 Nodes

  から成り，
  - エッジ e には始点 start(e) と
    終点 terminal(e) がある

  ことを表そうとしている



具体的なグラフの例

- 具体的に，Edges と Nodes に集合が割り当てられ，射 start と terminal に関数 Edges→Nodes が割り当てられると**具体的なグラフ**が1つ決まる

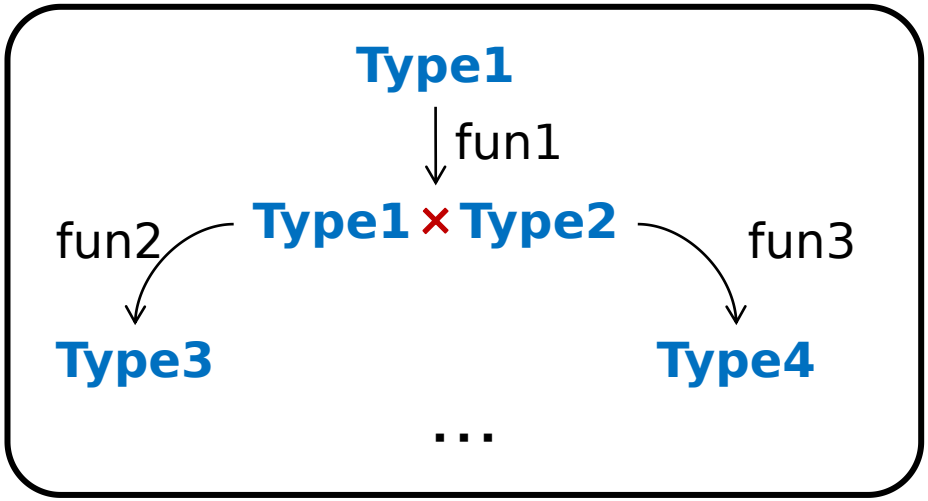
圏の例：プログラミング言語の圏
非常に簡単な関数型言語の圏

- **プログラミング言語**を**圏とみる見方は色々ある**と思うが，次のようなオブジェクトと射の圏が考えられる

  - **オブジェクト＝データ型**
    通常，**基本データ型**から C 言語でいう構造体 struct {T1 x; T2 y;} などの**型構築子**を使って無数の型を作る仕掛けが用意される

  - **射＝プログラム（関数）**
    通常，基本的な関数があり，それらを組み合わせて，さらに複雑な関数を構築するための仕掛けが用意される

プログラミング言語の圏のイメージ

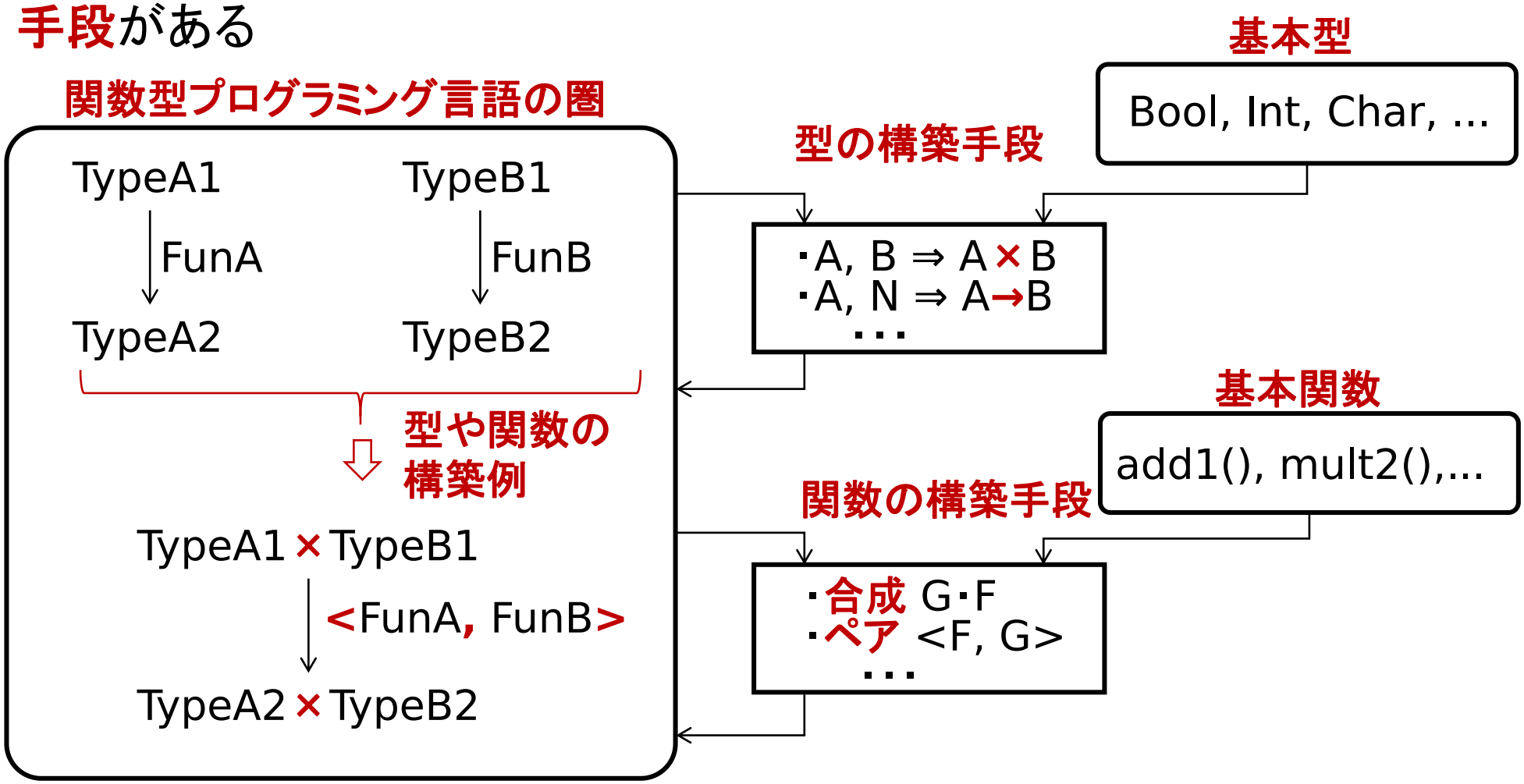


射の合成規則の例
$fun3 \cdot fun1 := \lambda x.fun3(fun1(x))$

注意）ここでは，「型や関数の実体が何か？」はまだ考えなくてよい．
ある文法に従った文字列と思っておいてよい．

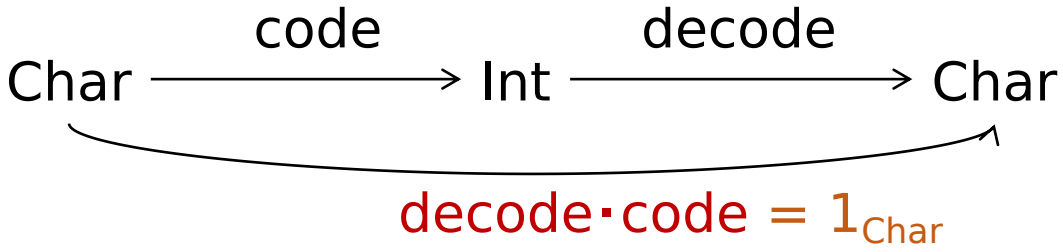
圏の例：プログラミング言語の圏
非常に簡単な関数型言語の圏（全体像）

- 「型」（オブジェクト）も「関数」（射）も**基本のもの**から**新たなものを作る手段**がある



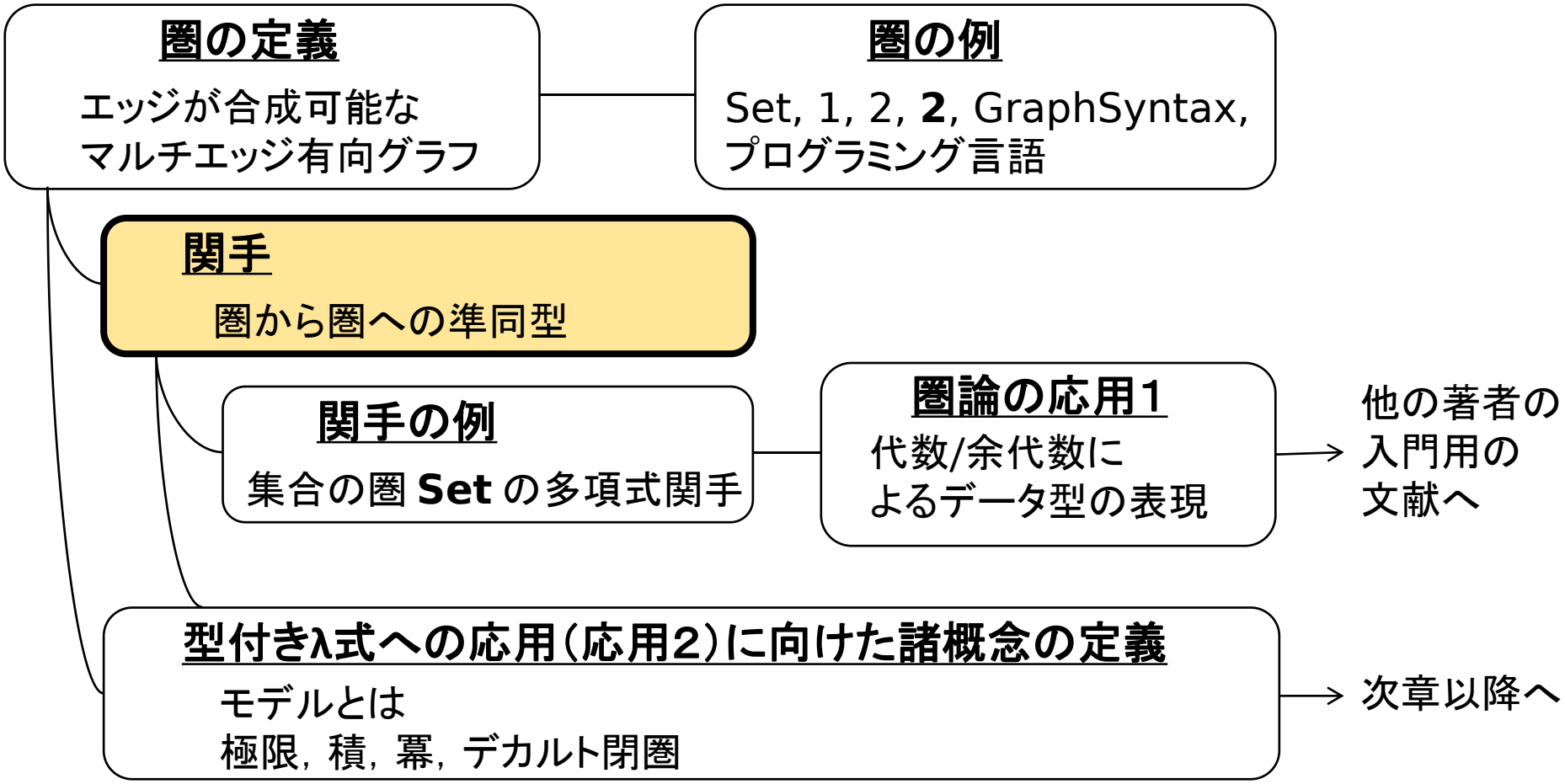
圏の例：プログラミング言語の圏
いろいろなレベルの抽象化

- 前頁で述べたプログラミング言語の圏は，まだ**「意味」が入っていない**
- 例えば，次のような型と射があったとする
  - 型
    - Int … 整数型
    - Char … 英文字の型
  - 射
    - code : Char → Int
    - decode : Int → Char
- このとき次のように合成された射は 1_{Char} であるべきである



- このような射を**等しいと見なす**ことによって，**意味**をより反映した圏を作ることができ，これには色々なレベルの反映の仕方がある

関手


圏の定義
エッジが合成可能な
マルチエッジ有向グラフ
圏の例
Set, 1, 2, 2, GraphSyntax,
プログラミング言語
関手
圏から圏への準同型
関手の例
集合の圏 Set の多項式関手
圏論の応用1
代数/余代数に
よるデータ型の表現
他の著者の
入門用の
文献へ
型付きλ式への応用(応用2)に向けた諸概念の定義
モデルとは
極限，積，冪，デカルト閉圏
次章以降へ

関手の定義

- **定義**　関手（functor）　圏の間の準同型

  C, **D** を圏とするとき，**C** のオブジェクトから **D** のオブジェクトへの対応 F_0 と **C**の射から**D**の射への対応 F_1

$$\begin{pmatrix} F_0 : \mathbf{C_0} \to \mathbf{D_0} \\ F_1 : \mathbf{C_1} \to \mathbf{D_1} \end{pmatrix}$$

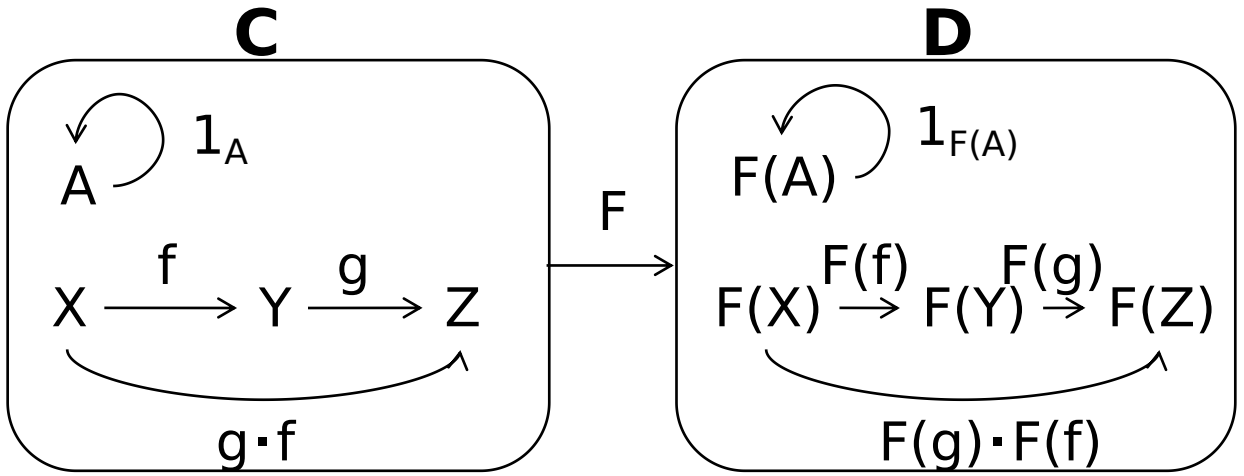
  の組が以下の条件を満たす時，これらの組 $F=(F_0, F_1)$ を **C** から **D** への**関手（functor）**という．記号では $F : \mathbf{C} \to \mathbf{D}$ と書く．また，正確には，オブジェクトの対応は $F_0(x)$，射の対応は $F_1(a)$ だが，どちらも F を使って F(x), F(a) と書く．また，さらに括弧を省略して Fx, Fa と書くこともある．

条件1（1を1に移す）

$F(1_A)=1_{F(A)}$

条件2（合成を保つ）

$F(g \cdot f)=F(g) \cdot F(f)$



関手の例
集合の圏 **Set** の上の多項式関手

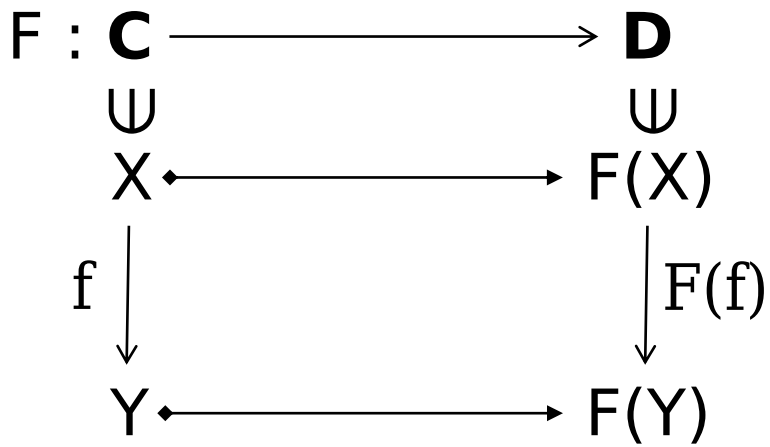
- 圏から圏への**関手は非常に多彩**である．ここではまず，**特殊**ではあるが，計算機科学への重要な応用を持つ集合の圏 **Set** から **Set** への**多項式関手（polynomial functor）**の例を紹介する
  - 定数関手
  - 恒等関手
  - 定数との直積の関手
  - X, X^2, X^3, …
  - 直和の関手
  - **多項式関手（polynomial functor）**

多項式関手を用いた代数と余代数によるデータ型の規定

（これらは**集合の圏に限った話ではない**が，親しみがあり，新たな概念（積オブジェクト，和オブジェクト）を導入する必要のない集合の圏で説明する）

関手の定義の表し方

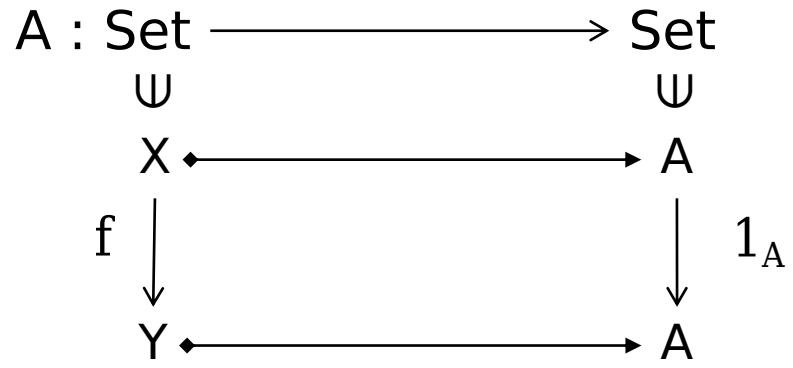
- 関手は**オブジェクトの対応**だけではなく，**射の対応**も規定する
- したがって，**2つのオブジェクトと一つの射**を含むような，つぎの書き方で定義する関手をうまく表現できるだろう



要素同士の対応を表すことにする．
本当は，出発点が縦棒の矢印「↦」を使いたかったのだが，使っているプレゼンテーション作成ツールの線に適当なものが無かったため代用した．

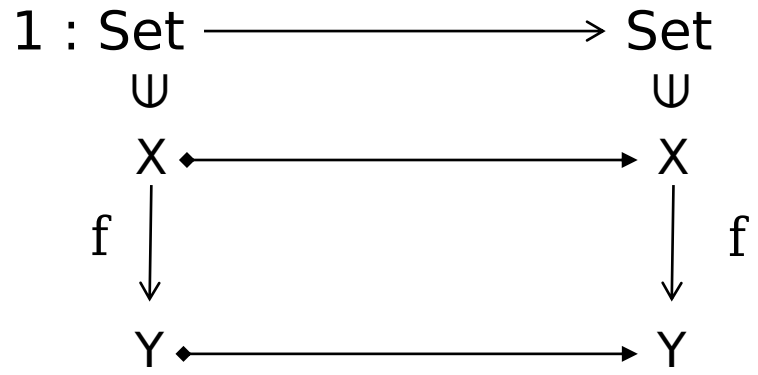
多項式関手
定数関手と恒等関手

- **定数関手 X↦A**



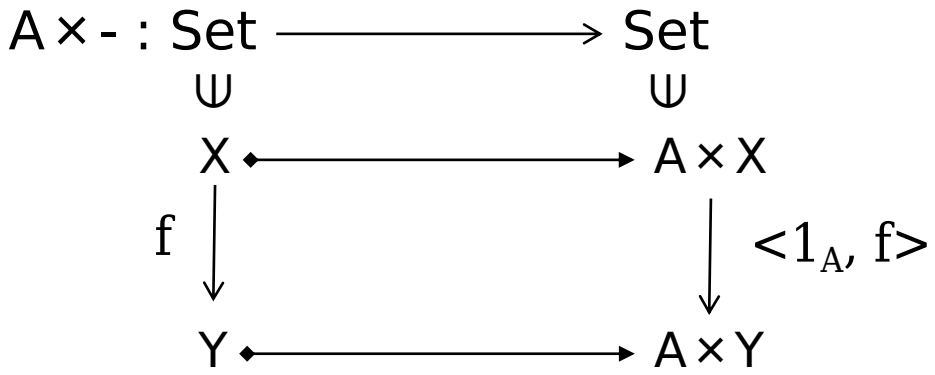
例） X ↦ {0}

- **恒等関手 X↦X**

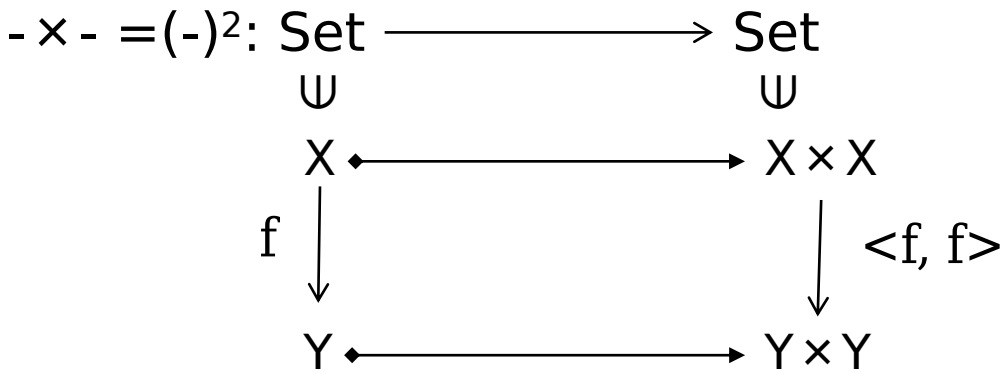


多項式関手

- **直積による関手**
  - $X \mapsto A \times X$　　. . . ある決まった集合 A と直積をとる関手

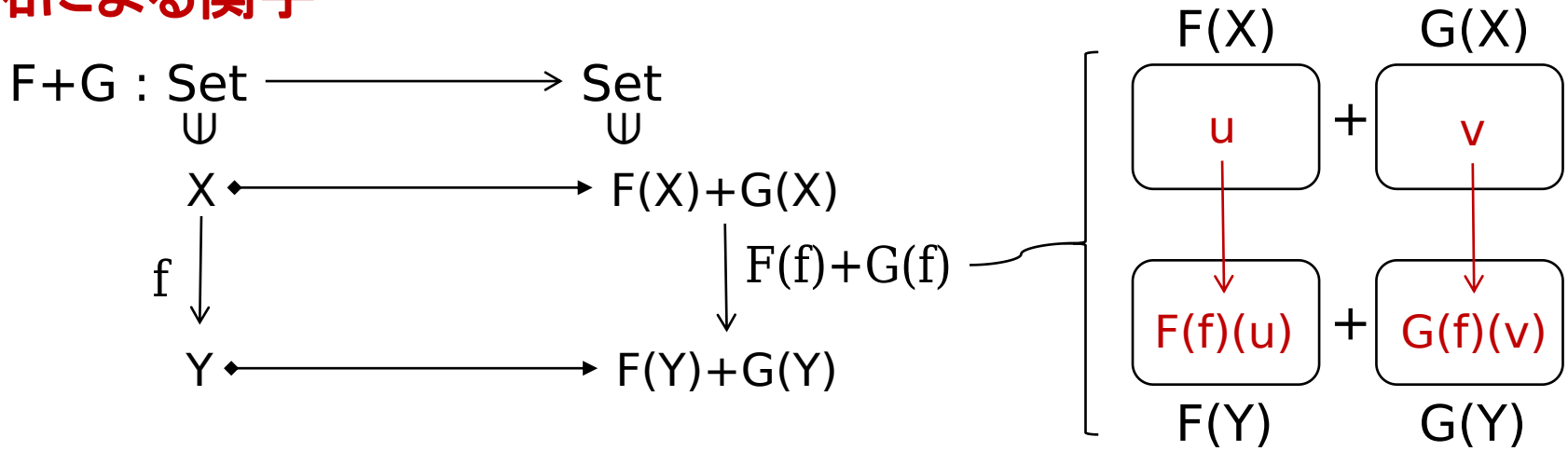


  - $X \mapsto X,\ X \mapsto X^2,\ \ldots,\ X \mapsto X^n$



多項式関手

- **直和による関手**



- **多項式関手（polynomial functor）**
  - A+B×X + C×X×X+... のような有限項の和からなる関手
  - 形式的には定義を理解できると思うが，
    具体的には，多項式関手にどんな意味があるのか？

→ひとつには代数や余代数と関係がある

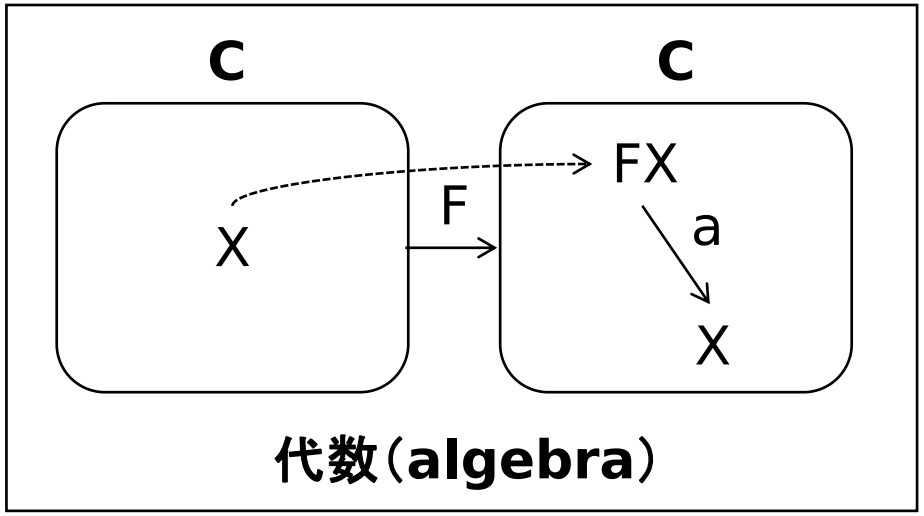
代数(Algebra)と余代数(Coalgebra)

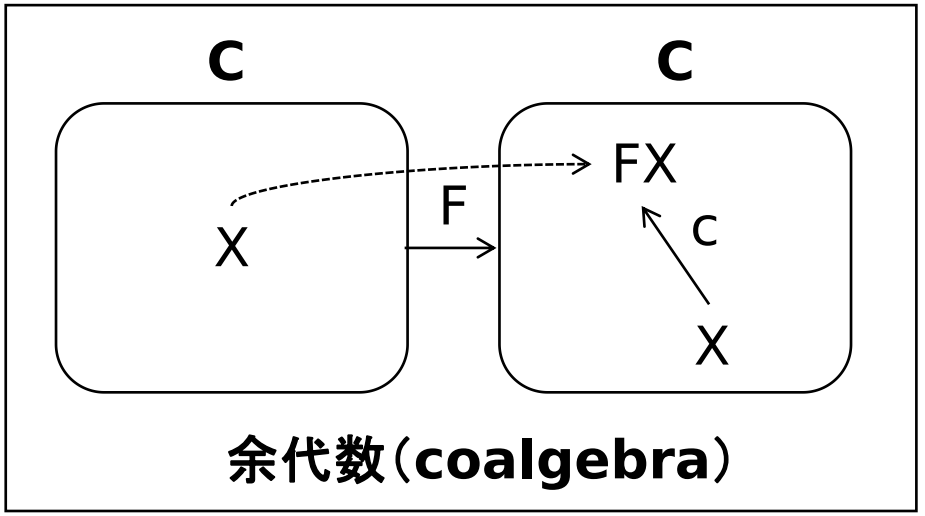
- **定義**
  - **C** を圏, F : **C** → **C** を自分自身への関手, X∈**C** とする. このとき, (気持ち的にはオブジェクト X や関手 F も含めて), 圏 **C** の中の

射　a : FX → X　を **代数(algebra)**

射　c : X → FX　を **余代数(coalgebra)**

と言う





代数(Algebra)と余代数(Coalgebra)

- 代数・余代数

$FX \xrightarrow{a} X$	$X \xrightarrow{c} FX$
代数	余代数

- これらは何を意味しているのか？
  - 関手 F : Set → Set （FX := A + X2 ） で考えてみる
  - このとき，例えば代数(つまり Set の中の写像)は

$$A + X^2 \xrightarrow{i + m} X$$

次のことを言っている

- X には **A の要素が埋め込まれている**(i)
- X の二つの要素 **x, y に対して，X の要素 m(x, y) を割り当てる規則**がある

  - つまり **A の要素から二項演算 m を使ってできる集合(代数)**を表している

代数(Algebra)と余代数(Coalgebra)
抽象データ型と代数

- **代数の例 list of integer**
  - 適当な架空のプログラミング言語を仮定して記載している

```
data_type list_of_integer
    empty : → list_of_integer
    cons : (integer, list_of_integer) → list_of_integer
end
```

  - これは,
    - A: {empty_list}
    - Integer : 整数の集合

  の定義のもとに, F=A + Integer×(-) : **Set** → **Set** という多項式関手の代数

$$F(D) \xrightarrow{<\text{empty, cons}>} D$$

  である

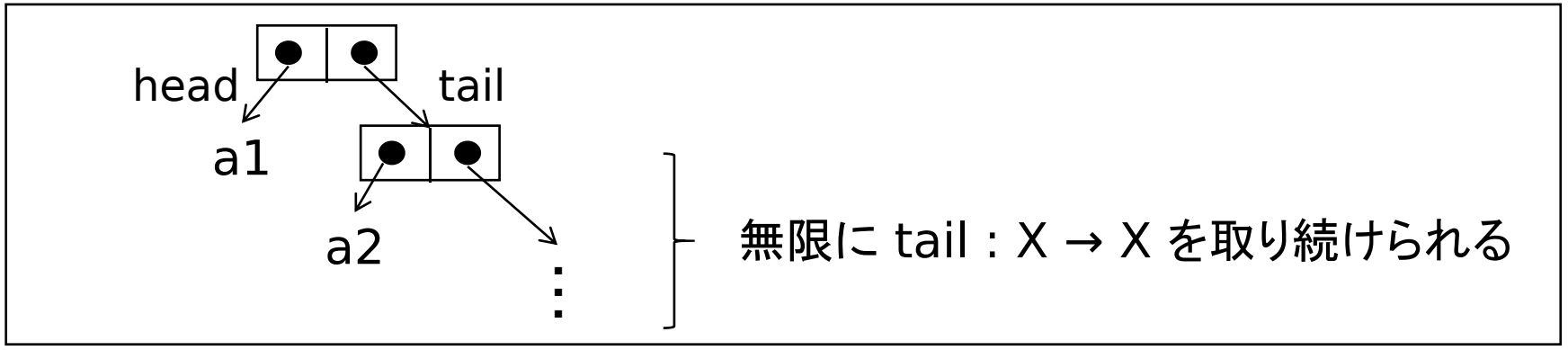
代数(Algebra)と余代数(Coalgebra)
計算機科学における余代数の例

- **余代数**の例 **infinite list of A**
  - **GX := A + X** で次のような余代数の例を考えてみる

$$X \xrightarrow{\text{head + tail}} GX=A + X$$

  - つまり，X は
    - A に属するデータを取り出す関数 **head : X → A** と
    - X の部分を取り出す関数 **tail : X → X** がある

ような領域(集合)であり，これは無限リストを表している



一般に余代数は，このような**無限のデータ構造**や**状態遷移機械**(無限に動き続け，何か信号を入れるとその都度反応を返す機械は無限リストと同じ)を表すのに利用できる

代数（Algebra）と余代数（Coalgebra）まとめと補足

- 多項式関手 F に対して，代数 FX→X，余代数 X→FX が一意に決まる訳ではない．
- これらを一意に決めるときには，それらが，**「始代数（initial algebra）」**や**「終代数（final algebra）」**であることを要求する必要がある
- 始代数や終代数で規定されたデータ型に関する**関数の定義方法**や**証明方法**として，**帰納法（induction）**，**余帰納法（coinduction）**がある
  - list_of_integer の長さ **length** の帰納法による定義
    （constructor に対する関数の挙動を指定することで関数を定義）
    - **length**(**empty**) = 0
    - **length**(**cons**(_, x)) = 1 + **length**(x)
  - infinite list of A のマップ関数 **map** の余帰納法による定義
    （関数に対する destructor の挙動を指定することで関数を定義）
    - **head**(**map**(list, f)) = f(**head**(x))
    - **tail**(**map**(list, f)) = **map**(**tail**(list), f)

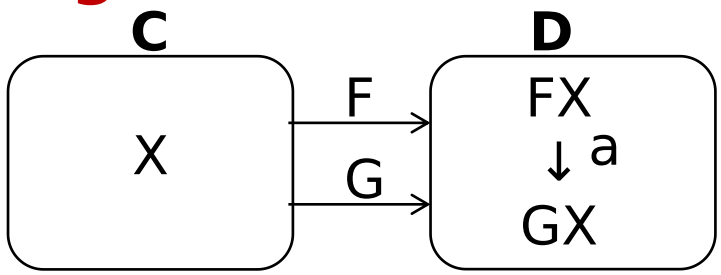
代数(Algebra)と余代数(Coalgebra)
読み物

- この代数，余代数の話題に関しては次の**秀悦なチュートリアル**がある．これはすでに今までの圏論の知識で読み始められる(と言うか，集合の圏に特化して圏論の最初から解説してあるのでいつでも読める)
  - Bart Jacobs , Jan Rutten : A Tutorial on (Co)Algebras and (Co)Induction, 1997, 38 pages

- 代数，余代数の両方の性質を持つ **dialgebra** というものもある

これは **C** → **D** の型の二つの
関手 F と G を使った

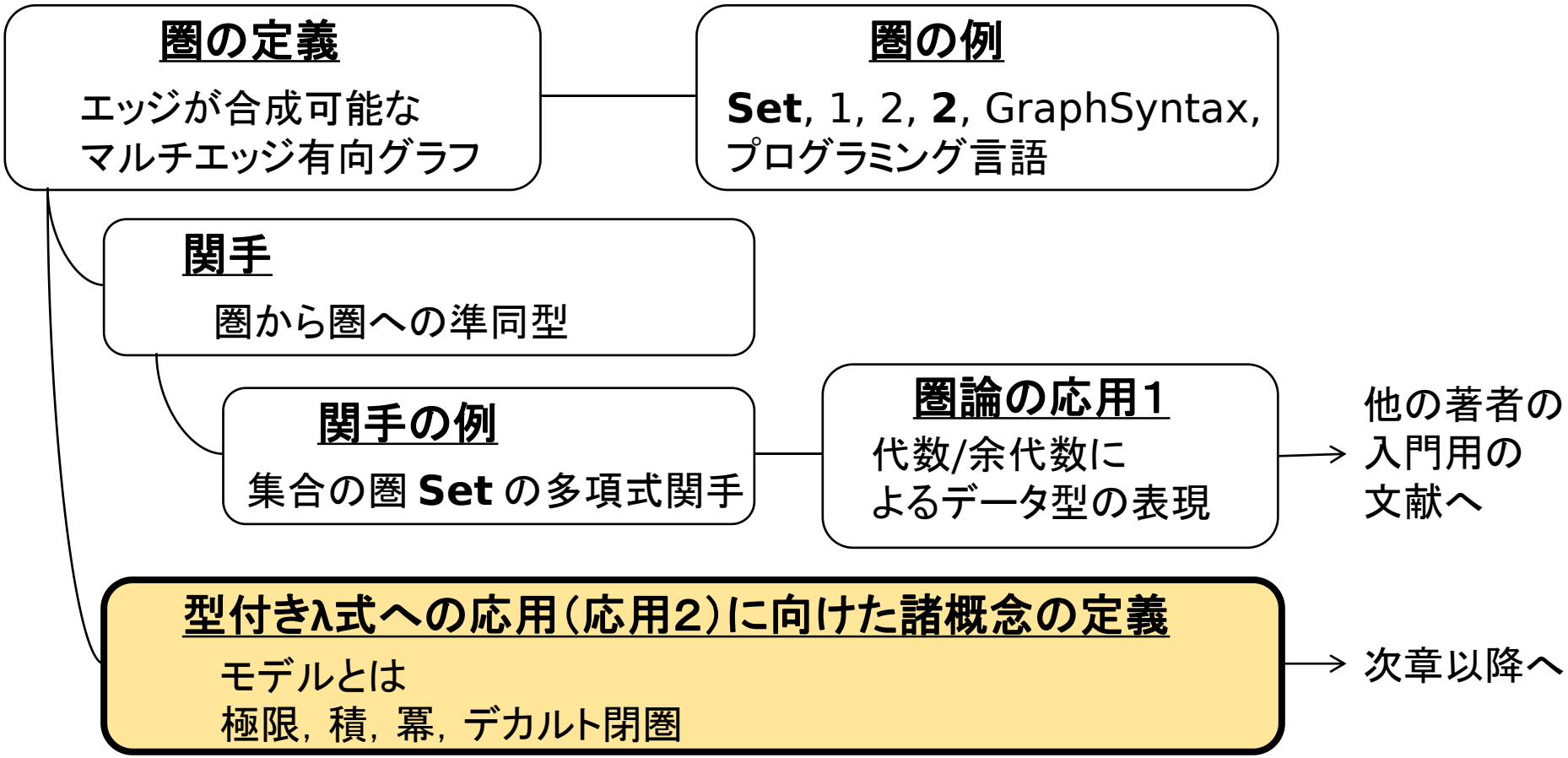
$$FX \xrightarrow{a} GX$$



の形の構成物で，萩野達也氏はこれや随伴関手を使っていろいろなデータ型を作ることができる言語(**CPL:Categorical Programming Language**)を提案した

- **萩野達也「カテゴリー理論的関数型プログラミング言語」1990年1月 等**

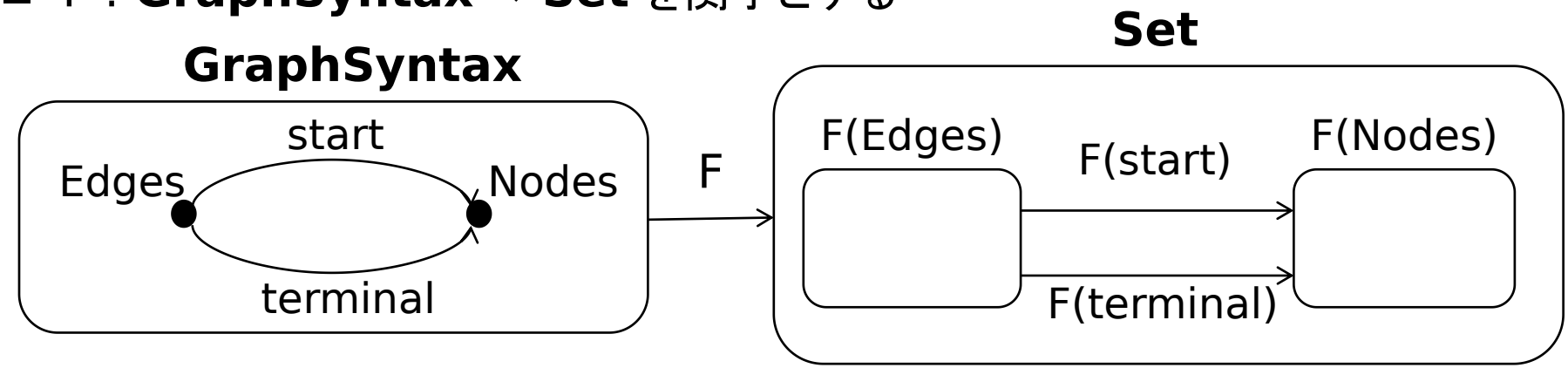
型付きλ式への応用（応用2）に向けた諸概念の定義



モデルとしての関手

- **定義**
  - 関手 F : **C** → **Set** を**圏 C のモデル**と言う
  - 関手 F : **C** → **D** を**圏 C の圏 D によるモデル**と言う

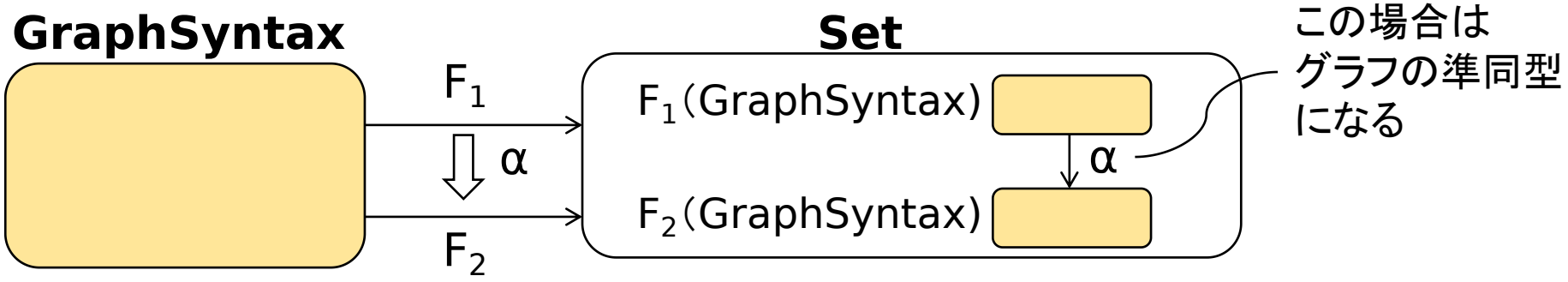
- 例 （圏の例で紹介した）圏 **GraphSyntax** のモデル
  - F : **GraphSyntax** → **Set** を関手とする



  - F は具体的に，Edges に**エッジの集合**を，Nodes の**頂点の集合**を与え，エッジにその**始まり（start）**と**終わり（terminal）**となる頂点を割り当てている
  - つまり，**関手 F は具体的なグラフを一つ定義している**のに他ならない

モデルとしての関手
脇道:「自然変換」との関係

- 関手 F_1 と F_2 がともに，GraphSyntax → Set の関手とすると，F_1，F_2 はそれぞれ具体的なグラフを決めている



  - 今日は触れないが，圏論では**関手から関手への射**として，**「自然変換（natural transformation）」**という概念を考える（上図のα）
  - このGraphSyntax の例の場合，**自然変換 α** は関手 F_1 が表す**グラフから**もう一つの関手 F_2 が表す**グラフへの準同型**になっている
- 圏論を勉強するときは**「自然変換」はとても重要**な概念である
- ここで述べた，自然変換が**モデルの間の準同型**になっているということは，**自然変換をイメージする**のに非常に役に立つと思う

意味を定めるための圏
特にプログラミング言語を解釈するための圏

- 集合の圏などへの**関手がモデルを定める**ことを見てきた
  - F_1 : **C** → **Set**　　**C** で規定される概念を集合で解釈
  - F_2 : **C** → **D**　　**C** で規定される概念を圏 D の言葉で解釈

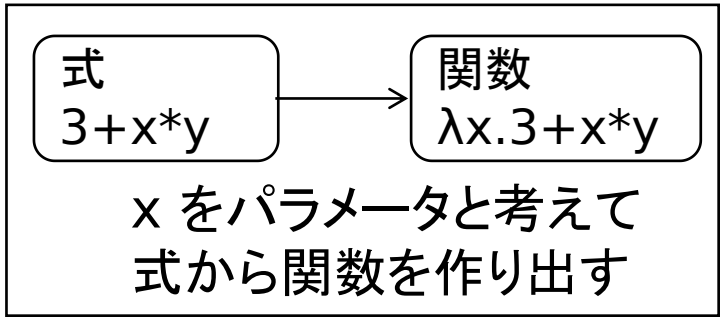
- 圏論の計算機科学への**応用の一つ**として，このモデルを定めるということを使い，**プログラミング言語などの意味論**を与えるというテーマがある

- 今日の発表はこのテーマに沿って，**単純型付きλ計算**の**意味論**を与える圏（デカルト閉圏）を構成する（**Set** もこのような圏だが，ここでは **Set** 以外の圏も考える）

- まずは，**プログラミング言語を埋め込む**ことができる圏が**どのような特徴を持っていればよいか**を見ていく

単純型付きλ計算の意味を与える圏の条件

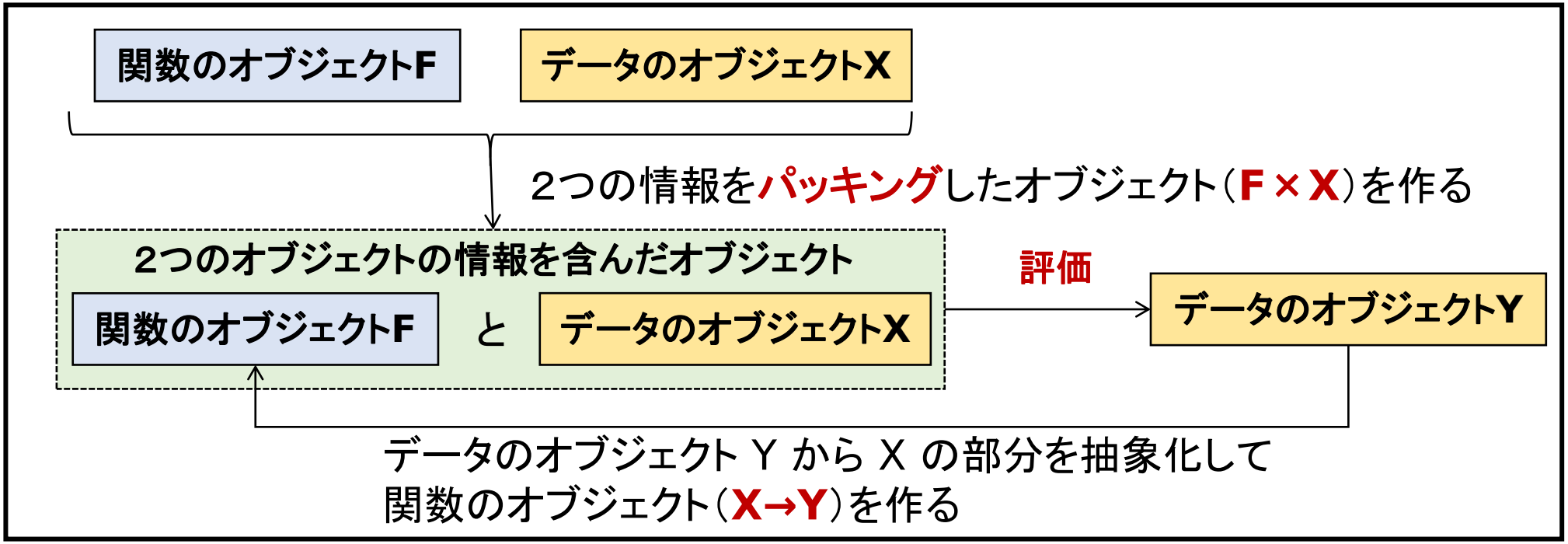
- 少し先走って，**「単純型付きλ計算」の特徴**を説明する．
- 「単純型付きλ計算」は次の2つの操作を持つ言語である
  - **関数を式に適用する**
    - f を式 3 に適用する (f 3)
  - **式から変数を抽象化して関数を作り出す**
    - 式 3+x*y の x をパラメータと考えた関数 λx.(3+x*y) を作り出す
    - 上の関数適用の例で f と書いていたものの実際の形は，このようなλx を伴った式である



- 「単純型付きλ計算」のモデルとなる圏は，これらの操作，**関数適用と関数への抽象化**に対応する**機構**を内部に持たねばならない
- 以下，どんな**機構**を持たねばならないかを説明する．それらを持つという条件が，**デカルト閉圏（CCC）**の条件になる

単純型付きλ計算の意味を与える圏の条件

- 圏における**関数適用**と**式から関数への抽象化**を表現する機構とは？
- つぎの3つの機構でこれらを表現する
  - 二つのオブジェクトの情報を一つのオブジェクトへの**パッキング**
  - パッキングされたオブジェクトの**評価**
  - オブジェクト Y からオブジェクトX の部分を抽象化して**関数のオブジェクト F** を作る



単純型付きλ計算の意味を与える圏の条件
積オブジェクトと冪オブジェクトを持つこと

- これも先走って言うと，前頁の図を実現するための圏の概念は**「積」**と**「冪」**で，集合の圏 **Set** での対応物は次のようになる

名称	記号	直感的な説明	集合の圏での対応物
積（Product）	A×B	2つのオブジェクトA と B の情報を**パッキング**したオブジェクト	A×B（A と B の直積）
冪（Exponential）	A→B（あるいはB^A）	データのオブジェクト B から A の部分を抽象化した**関数のオブジェクト**	A→B（A から B への関数の集合）

- 以下，**Set** 以外の一般の圏において，これらのモノを定義していく

積オブジェクト
準備：Generalized Elements

- 二つのオブジェクト A と B の両方の**「情報（あるいはデータ）を含んだ」「積オブジェクト（product）」**を作りたい

- オブジェクトの**「情報（あるいはデータ）を含んだ」**とは何か？
  - 圏論では一般にオブジェクトの**内部の要素は見えない**が，それに代わる概念はある
  - それは射である
  - 特に集合の圏 Set で1個の要素の集合 {*} からの写像 f : {*} → A は，A の要素と同一視できる f(*)∈A

定義

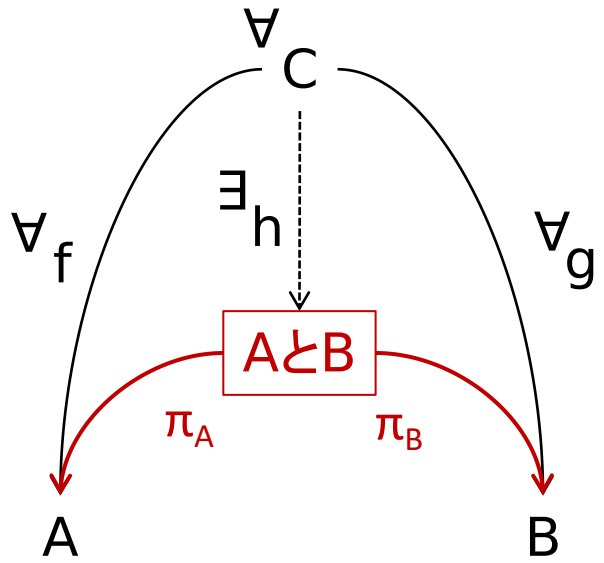
射 f: X → A を，A の generalized element という

- これは，射はオブジェクトの中の要素のように見えることもあるので，それを思い起こさせる**別名**をつけておこうというだけである
- 圏論では，「モノの中の要素は重要でない」と言いつつ，ここでは，「モノの持つ情報」を考える直感的な道具として「要素」のアナロジーを使わせていただく

二つのオブジェクトの積オブジェクト
二つのオブジェクトへの射の組合せを中継できるオブジェクト

- 2つのオブジェクト A と B の情報(データ)をすべて持っているオブジェクト**「A と B」**とは？
  - 任意のオブジェクト C に対して
    - Cから A への任意の射 f と
    - Cから B への任意の射 g

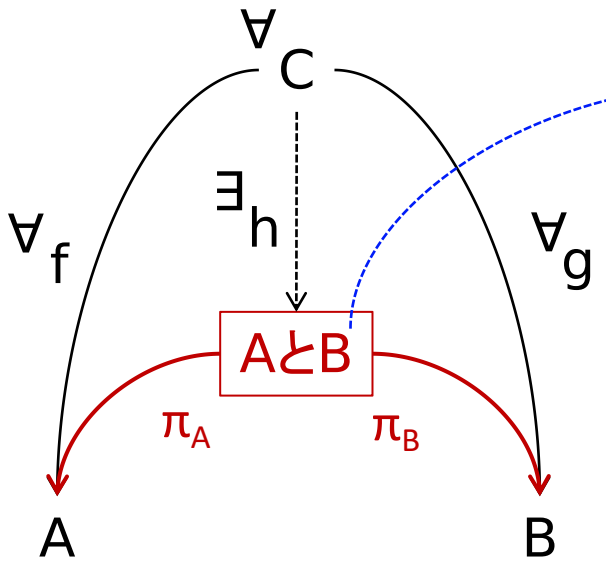
を中継することができるオブジェクト
「AとB」
は A と B の情報をすべて含んでいると考えてよいだろう！
なぜなら任意の generalized elements
f : C→A, g : C→B の組を中継する射
h : C→AとB があるので, f, g を考えるかわりに「AとB」のgeneralized element h を考えれば良いから



「AとB」
π_A : 「AとB」→A
π_B : 「AとB」→B
は固定

二つのオブジェクトの積オブジェクト
中継能力のあるオブジェクトの中の最小のオブジェクト

- **「A と B」**は，**f, g を中継する射 h が存在**するだけの条件で良いか？
  - 集合の圏で考えると，集合 A と B の直積 A×B = {(a, b) | a∈A, b∈B} は，このような h の存在を保証するが，じつは，このような保証は，**A×B より大きな集合ならなんでも可能である**（すべての (a, b) の組を含んでいれば良い）



A×B, {0, 1}×A×B など，A×B より大きければ，どんな集合でも，この「AとB]の条件を満たす．

通常は，最小性の条件を課す（次頁）

（＝h が一意に決まるという条件）

二つのオブジェクトの積オブジェクト
改めて標準的な定義

- **定義** オブジェクト A と B の積オブジェクト（product）
  - オブジェクト A と B に対して，オブジェクト A×B と二つの射 π_A と π_B

$$A \xleftarrow{\pi_A} A \times B \xrightarrow{\pi_B} B$$

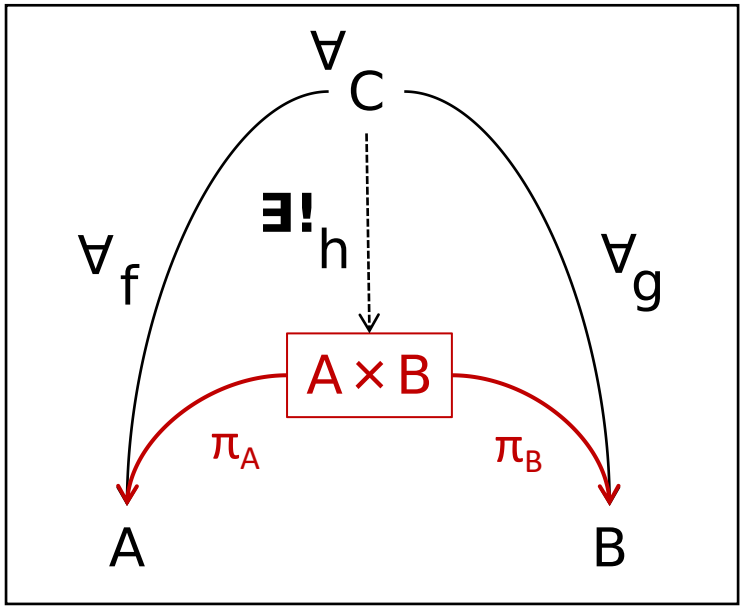
が，オブジェクトAとBの**積オブジェクト（product）**であるとは，

- 任意のオブジェクト C と
- Cから A への任意の射 f と
- Cから B への任意の射 g に対して

$f = \pi_A \cdot h$

$g = \pi_B \cdot h$

となる射 $h : C \to A \times B$ が**ただ一つ存在**することである．このように図式の中の射の合成が等しくなることを**「図式を可換にする」**という．

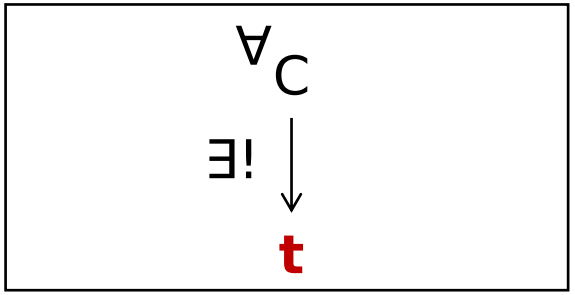


注意）∃! あるいは単に ! は**「ただ一つだけ存在する」**ことを表す記号である

終対象：0個のオブジェクトの積オブジェクト

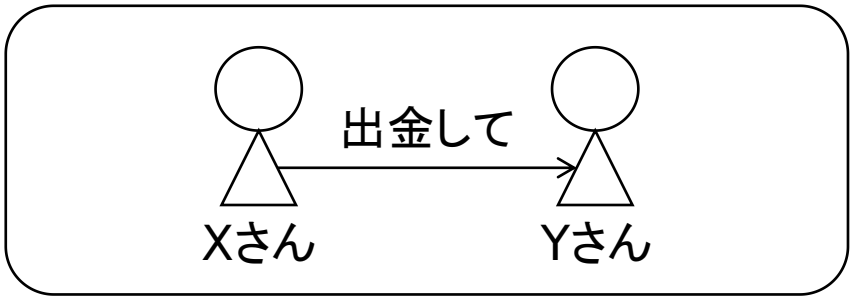
- ここまで**2個のオブジェクトの積オブジェクト**を定義してきた
- これは一般に **n≧1 個のオブジェクトの積オブジェクトへと拡張**できる
  - つまり n 個のオブジェクトへの任意の射の組合せを中継できる1個のオブジェクトという具合に
- また，逆にオブジェクトの数を減らして，**0 個にすることも**考えられる
  - つまり，「どのオブジェクトからの射を中継する必要がある」という制約が全く無く，**すべてのオブジェクトから1個だけの射を持つオブジェクト**である．このようなオブジェクトを**終対象**という．終対象は，**t** や **1** で表すことが多い．

- **定義**　終対象
  - 圏の中の任意のオブジェクトから
    ただ1つの射があるオブジェクトを
    終対象（**terminal object**, **final object**）と言う

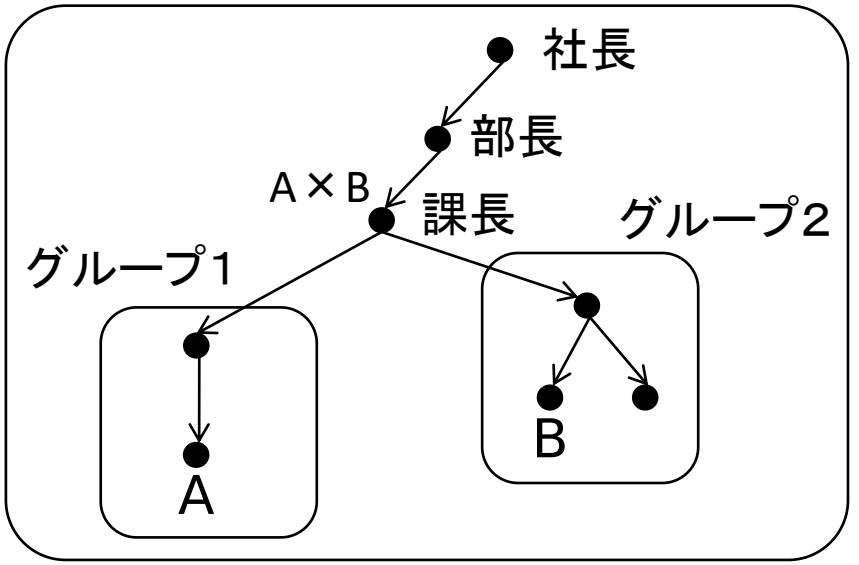


比喩：組織の中の積オブジェクト

- 次のような **z社の圏**を考えてみる
  - オブジェクト
    - 社員
  - 射
    - 社員から社員への指示
    - 合成は適当に決めて！



- 積オブジェクトＡ × Ｂは，ＡさんとＢさんへの指示を中継できる最も職位の低い人，つまり，**Ａさんと Ｂさんの共通の上長で最も職位の低い人**である．



こういう**比喩はいくらでも**できる．

また，**射の向きを逆にすると概念ががらっと変わる**こともあるので，そのことに**注意しながら**，いくつか味わってみることも面白い．

補足：積オブジェクトと「極限」との関係

- 通常の圏論のテキストでは，**積オブジェクト**は**「極限」**の一種としての統一的な説明がなされる
- 本資料では，できるだけ速やかに積オブジェクトと冪オブジェクトを定義するため，この扱いを避けた
- **「極限」**，およびその双対（射が反対向き）の概念である**「余極限」**の理解のためには，**「終対象」**，**「始対象」**の理解が基本的である
  - **終対象**についてはすでに説明した．オブジェクト **1** が**終対象**であるというのは，C の任意のオブジェクト X から，1 にただ一つの射があること．つまり，

$$\forall X \xrightarrow{!_X} \mathbf{1}$$

  - **始対象 0** はこの逆，つまり任意のオブジェクトにただ一つの射がある

$$\mathbf{0} \xrightarrow{!_X} \forall X$$

- オブジェクト A と B の積 A×B は，大まかに言えば，**「A と B の両方にアクセスできるオブジェクトがなす圏の終対象」**である（次頁）

補足：積オブジェクトと「極限」との関係

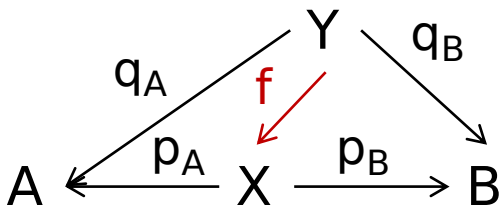
- A×B：A と B の両方にアクセスできるオブジェクトがなす圏の終対象
  - 圏 **C** から次のようにして，**圏 C(A, B)** を作る
  - **圏 C(A, B)**
    - **オブジェクト**

      X∈**C** で，A にも B にも射があるものに対して，X とそれらの射を組にしたもの

$$A \xleftarrow{p_A} X \xrightarrow{p_B} B$$

    - **射**

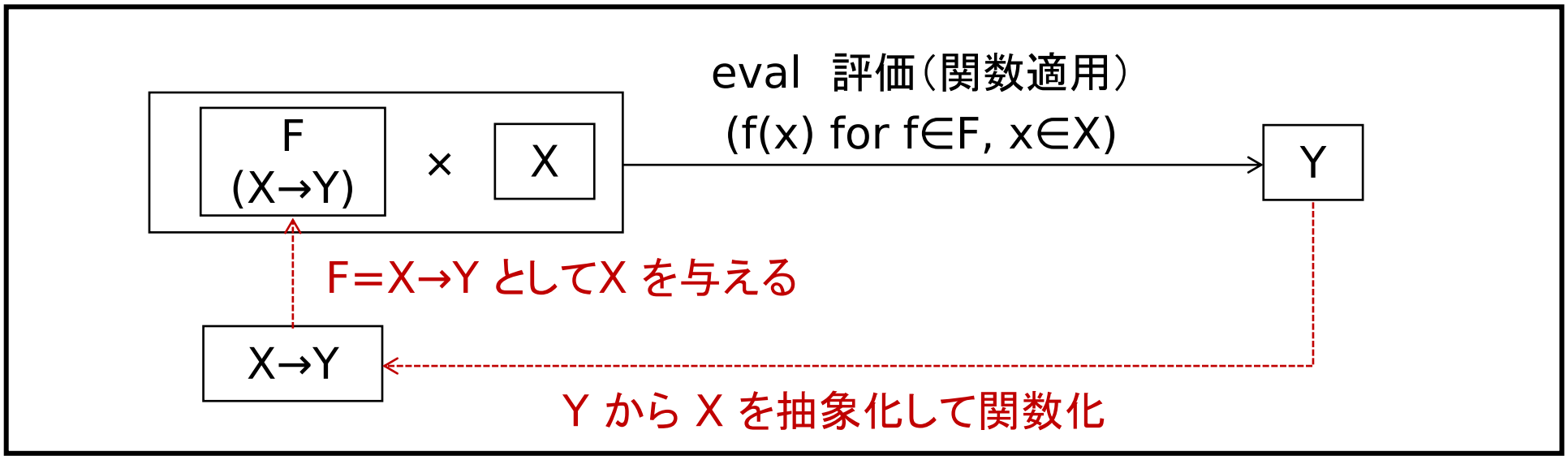
      $q_A = p_A \cdot f$ かつ $q_B = p_B \cdot f$ となる
      f をオブジェクト (Y, q_A, q_B) から
      (X, p_A, p_B) への射とし，射の合成は
      圏 **C** の中の射の合成のままとする．



- このようにして作られた**圏 C(A, B) の終対象が A×B** である．意味的には A, B にアクセス可能（射がある）オブジェクトからの射を中継できるオブジェクトの中の最小のものである

評価射(eval)と冪オブジェクト(Exponentials)

- 前頁までで，二つのオブジェクトの**積 F × X** が定義できた
- 次に**関数適用**(eval: (F×X)→Y)と**関数への抽象化**(X→Y)を定義したい



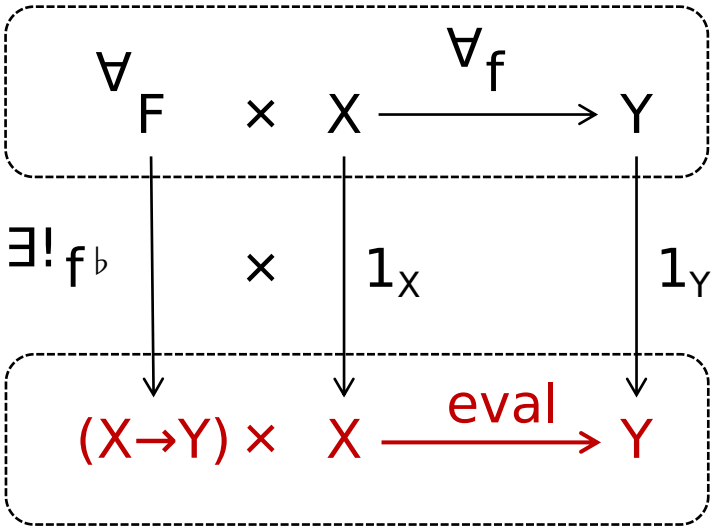
- **(X→Y)×X $\overset{\text{eval}}{\rightarrow}$ Y はどのような性質を持っていれば良い**だろうか？

  X と Y に対してこのような射 eval がありさえすれば良いのだろうか？

冪オブジェクトと評価射

関数空間の中の関数空間，インタープリタの中のインタープリタ

- X→Y を単に**「射 f：F×X → Y が存在する F」**とすると，かなりつまらないオブジェクトが X→Y になり得る
  - 例えば，集合では F=Y，$f=\pi_Y$ と置けば，π_Y：Y×X→Y の形になる
- 関数空間相当の X→Y としては，これらの**評価用の射をすべて取り込み得る**オブジェクトの中の**最も小さいもの**ととるべきだろう
- これは積オブジェクトの時と同じように，次のような条件を置けばよい

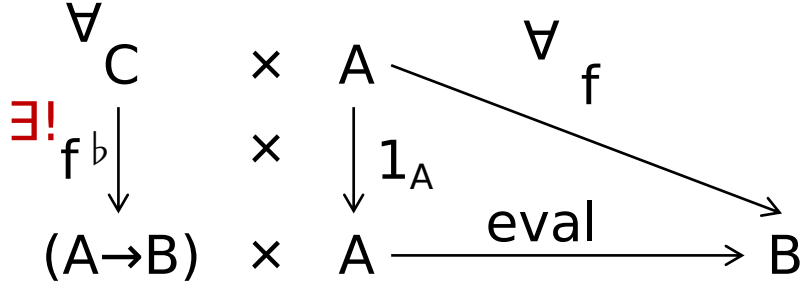


任意の関数適用の形式 f：F×X→Y に対して，
$f^\flat$：F→(X→Y) がただ1つ存在して
eval･($f^\flat$, 1X)=f
となる．
つまり，

- 任意の関数適用形式 f：F×X→Y を eval：(X→Y)×X→Y が**シミュレートできる**．
- ヨ！$f^\flat$ の部分から，それは**最小**のオブジェクトである

冪オブジェクトの定義
標準的な教科書の形式

- **定義**　**A と B の冪（exponential）** A→B　（B^A が普通）
  - **C** は任意の二つのオブジェクトの積が存在する圏とし，A, B, C を **C** のオブジェクトとする
  - **C** のオブジェクト A→B が（射 eval : (A→B)×A→B を伴って），A と B の**冪オブジェクト（exponential）**であるとは，次の条件が成立することである．
    - 任意のオブジェクト C と任意の射 f : C×A→B に対して，次の式が成り立つ射 $f^\flat$:C→ (A→B) が一意に存在する



左の図式が可換になる $f^\flat$，つまり

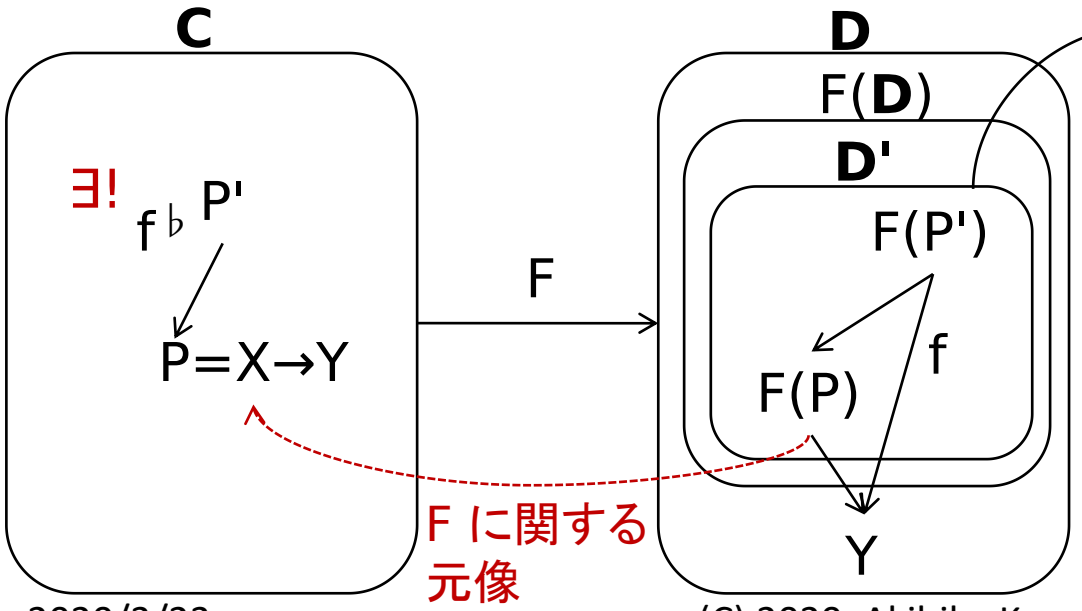
eval･<$f^\flat$, 1A>=f

になる $f^\flat$ が一意に存在する

注意）オブジェクト (A→B) は **B^A** が普通の記法である．
私は関数空間っぽい (A→B) を使うことが多い．

冪オブジェクトの統一的な理解に向けて

- 圏の多くの概念が**極限と余極限で整理**できるが，冪は直接には，これらで整理出来ず，通常は，**随伴関手の概念の登場**を待つ必要がある
- しかし，もう少し簡単な理解方法として，冪は**「陰関数型の極限」**，あるいは**「逆関数型の極限」**とみる方法もある
  - F : **C** → **D** を関手とする（冪の理解の時は F(C) := C×X と置く）
  - 冪 P=X→Y は圏 D' := {Z∈F(C) | 射 F(Z)→Y が存在} の終対象（極限）の F に関する元像である



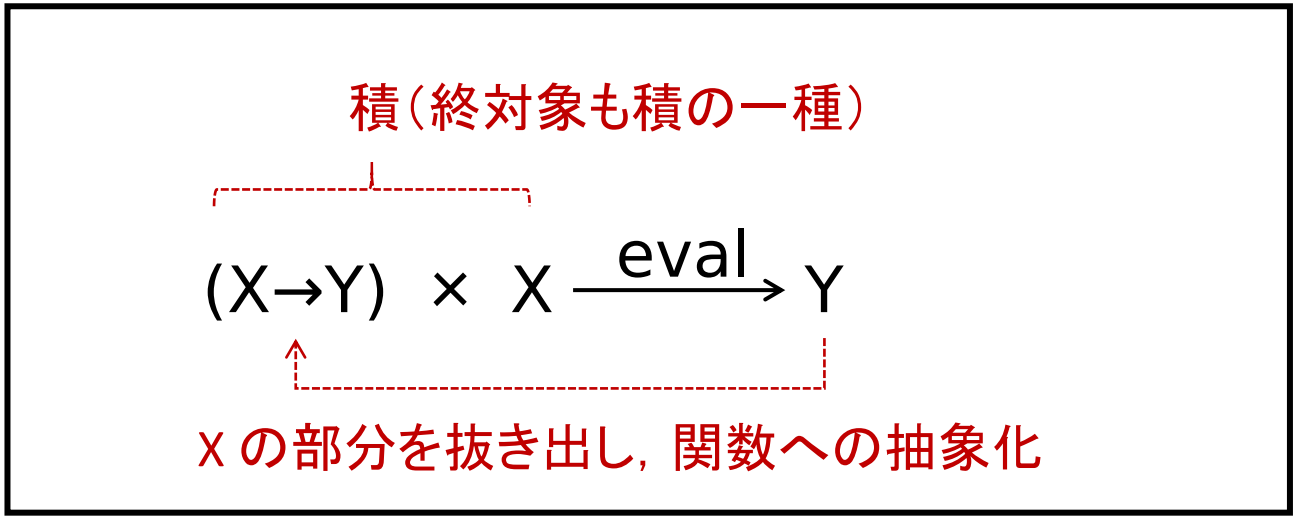
D の部分圏 **D'**
オブジェクト：F(D) の中で
Y への射があるもの

これは単に「冪が**コンマ圏**」
((-)×X ↓ Y) の終対象であると
言っているにすぎないのだが，
コンマ圏自体がイメージし
にくかったりする．

デカルト閉圏
Cartesian Closed Category CCC

- **定義**
  - 任意のオブジェクト X と Y に対して，**積 X × Y**，**冪オブジェクト X → Y** が存在し，さらに**終対象**が存在する圏を**デカルト閉圏**（**Cartesian closed category**）という．デカルト閉圏は英語の頭文字をとって，**CCC** と略されることもある．
  - デカルト閉圏は**関数適用**と**関数への抽象化**を表現する機構が入った圏である

デカルト閉圏 **C**



デカルト閉圏の例

- **集合の圏 Set**
  - $A \times B = \{(a, b) \mid a \in A, b \in B\}$
  - $A \to B = \{f \mid f$ は A から B への関数$\}$
    - $eval : (A \to B) \times A \to B,\ eval(f, x) = f(x)$
  - 終対象 1 は一つの要素だけからなる任意の集合 $\{*\}$

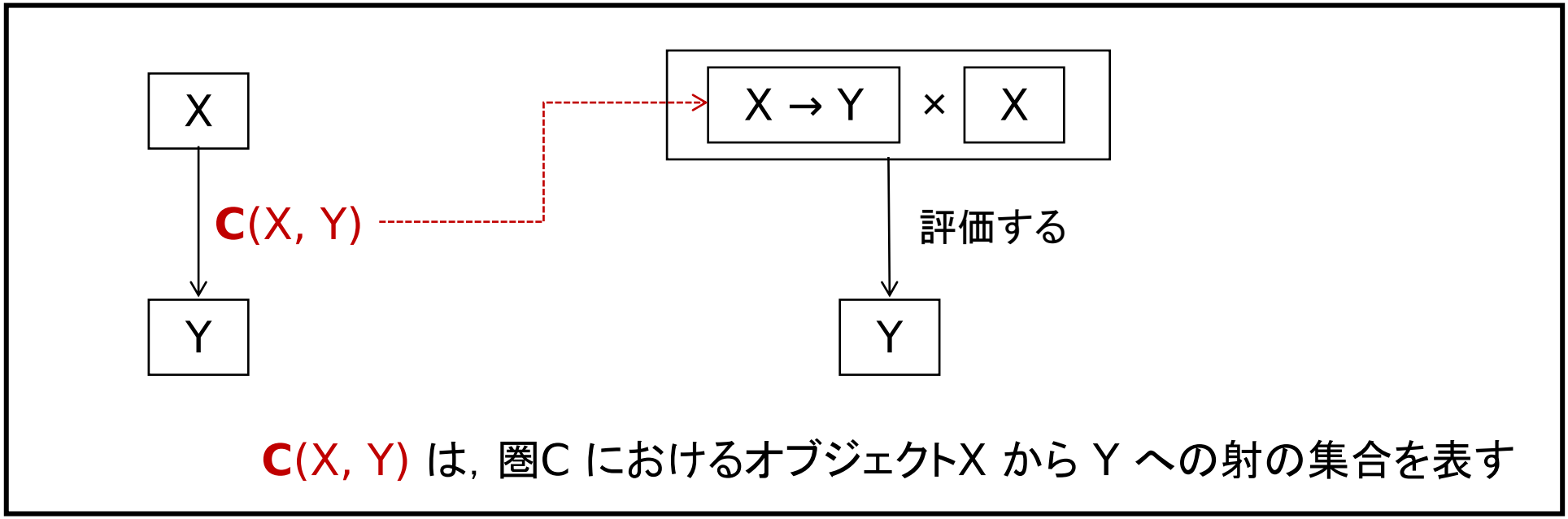
- **部分順序集合の圏 Poset**
  - 圏の定義
    - オブジェクト
      - 部分順序集合 $(P, \leq)$
    - 射
      - 順序を保つ関数 $f : P \to Q$ such that if $p1 \leq p2$ then $f(p1) \leq f(p2)$
  - $P \times Q = \{(p, q) \mid p \in P, q \in Q\}$ $(p1, q1) \leq (p2, q2)$ iff $(p1 \leq p2$ & $q1 \leq q2)$
  - $P \to Q = \{f \mid f$ は PからQへの順序を保つ関数$\}$ $f1 \leq f2$ iff $f1(p) \leq f2(p)$ $\forall p \in P$
  - 終対象 1 は一つの要素だけからなる集合 $\{*\}$, $* \leq *$

デカルト閉圏の例

- **Heyting Algebra**
  - 次のような条件を満たす poset (H, ≤)
    - 任意の2元 x, y∈H に対して下限 x∧y∈H が存在する
    - 任意の2元 x, y∈H に対して z∧x ≤ y となる z の上限が存在する. これをx→y と表す
    - H の最小元が存在する
  - poset H を圏と考えたとき, これは CCC になる
    - 圏 H
      - オブジェクトは H の元
      - 射 x→y は x≤y の時, かつそのときに限り x→y が唯一存在するとする
    - x × y は x∧y
    - x→y は上で定義した x→y
    - 終対象は H の最小元
  - Heyting Algebra は**直感主義命題論理のモデル**になる
  - これは CCC が**単純型付きλ計算**のモデルになるのと並んで, CCC の大切な用途の一つである

デカルト閉圏に関する補足

- デカルト閉圏は射の集合相当をオブジェクトとして扱える圏



- 領域理論で関数空間をデータ空間に埋め込める領域 D

$$[D \to D] \cong D$$

を構成する必要に迫られることがあるが，そのような領域の候補である

単純型付きλ計算

型なしλ計算と単純型付きλ計算

- まず，**型なしのλ計算**について簡単に説明する
  - ご参考：次の場所にλ計算の簡単な解説を書いた
    http://www.cs-study.com/koga/computer/lambda.html
    動かして遊ぶλ計算の初歩 in Ruby

- 次に，そのλ計算に，型による制限を加えた体系として**単純型付きλ計算**を導入する
- 簡単にいうと単純型付きλ計算は，**型**によって，λ計算に，**自己参照ができないような制限**を加えたものである．したがって，**再帰呼び出しができない（組込み型を導入することによる抜け道はある）**

型なしλ計算

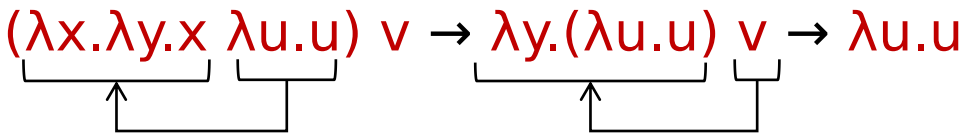
- **前提**
  - **変数**が可算無限個，用意されているとする（変数は好きなだけ使えるという意味）
- **λ式** … 次のように作られたものだけがλ式である
  - **変数**はλ式である
  - Exp がλ式で，x が変数なら **λx.Exp** はλ式である
  - Exp1, Exp2 がλ式なら，**(Exp1 Exp2)** はλ式である
- **例**: λx.x，λf.λx.(f x)，…
- **用語，記法**
  - λx.Exp という形があったとき，Exp 内に現れる変数 x は**束縛（bound）**されているという．また，変数（仮に u とする）はそれを取り囲むλu という形のものがないとき，**自由（free）**であるという

    例： λx.λy.(**z** λz.(**x** (**y z**)))

    左括弧の直後の **z** のみが自由で他は束縛されている
  - λ式の括弧は分かると思うときは，適宜省略する

型なしλ式の計算

- 次の2つの変換ルールを導入する
  - **α変換** … 変数(パラメータ)の名前変更
    - **λx.Exp** において，x を別の変数，例えば，y に変えて，**λy.Exp'** にしてよい．ただし，**Exp' は Exp の中の free な x をすべて y に変えたもの**である．また，Exp の中に **y は free に現れない**ものとする
  - **β変換** … 代入操作
    - λx.Exp1 Exp2 という形のλ式を **Exp1[x := Exp2]** に変えてよい．ただし，Exp1[x := Exp2] は，Exp1 の中の **free な x を Exp2 に置き換え**たものである．また，このとき，Exp2 の中の free な変数が束縛されるようなことがあってはならない
- λ式の計算
  - **α変換，β変換の繰り返しを計算の過程**とみなす
  - もうβ変換が適用できないλ式を**正規形(normal form)**であるといい，これを**計算の結果**とみなす
  - 例



いろいろなλ式とその計算能力(1)

- **自然数, 0, 1, 2, ... (チャーチ数)**
  - **$n = λf.λx.f(f(...f(x)...))**

f が n 個

注意)**$n** はこの資料だけの記法

  - $n は与えられた関数 f を n 回適用するという働きを持ったλ式である

- **真偽値(ブール値)**
  - **true = λx.λy.x**
  - **false = λx.λy.y**
    - true と false は, 次のような簡約が行われるλ式である. プログラミング言語の if 文「**if B then Exp1 else Exp2**」の B の位置にそれらがあると, それぞれ **then-part**, **else-part** が取り出される性質をもったλ式である
      - true Exp1 Exp2 => Exp1
      - false Exp1 Exp2 => Exp2
    - ちなみに false は $0 と同じ

いろいろなλ式とその計算能力(2)

- **不動点コンビネータ Y**
  - **Y= (λf . (λx . f (x x)) (λx . f (x x)))**
    = (λf . (λx . f (x x))
    (λx . f (x x)))
  - **YF→F(YF)→F(F(YF))→...** と無限に簡約化が進むλ式
    (つまり，指定されたFを何回も何回も繰り返し適用するλ式)
  - 再帰関数(あるいは**ループ相当の計算**)の表現に使う
  - 再帰関数を導入できる**不動点コンビネータは Y だけでなく無数にある**

- **λ計算の計算能力**
  - 数が表現でき，関数適用の連接，if 文，再帰(ループ)が表現できるので，**再帰関数**を表現することができる(**再帰的に枚挙可能な関数**も表現可能)

- 単純型付きλ計算の計算能力
  - 次に紹介する単純型付きλ計算はこれより少し計算能力が落ちる

単純型付きλ計算

- これから**単純型付きλ計算**を定義していく
- 基本的には型無しλ式の**λx.・・・の部分の変数 x にλx:τ のように型をつけただけ**だが，1つ**扱いで困難な点**がある
- それは，λで導入されない**自由変数を扱う必要**が発生し，λ式単独で意味を与えることが難しいということである
  - 例えば，**λx:τ.λy:σ.x** は **τ→(σ→τ)** と型が決まるが，中の式の**λy:σ.x** は x の型が不定のため単独では**型を決めることができない**.
  - 自由変数を持たない式だけ扱うようにすれば良いと考えるかもしれないが，後者の式は前者の式を解釈する途中に現れてしまう
- そのため自由変数の**型の宣言とλ式の対**に意味を与える必要がある

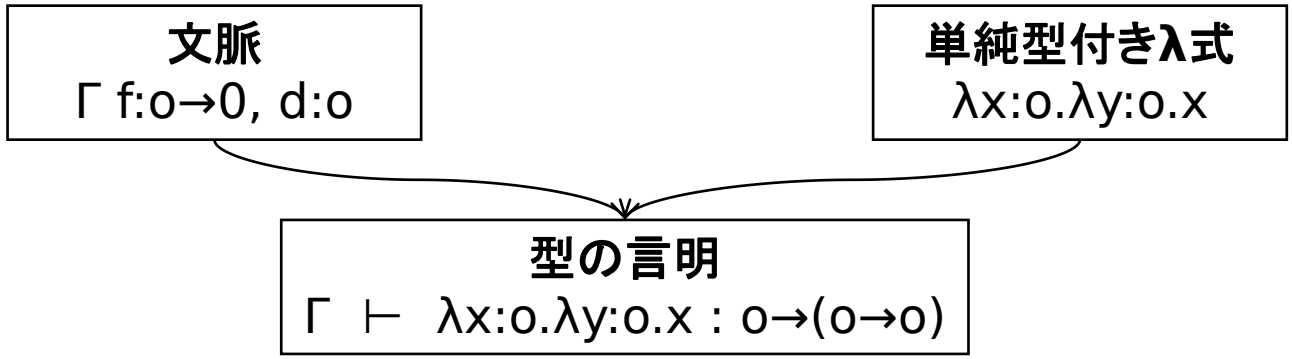
自由変数とその型の宣言の列　⊢　型付きλ式

例：　x : τ ⊢ λy:σ.x　　（x の型が τ である文脈での型付きλ式 λy:σ.x）

⊢ の左を**文脈（context）**という

単純型付きλ計算
この資料での記述方法についての注意

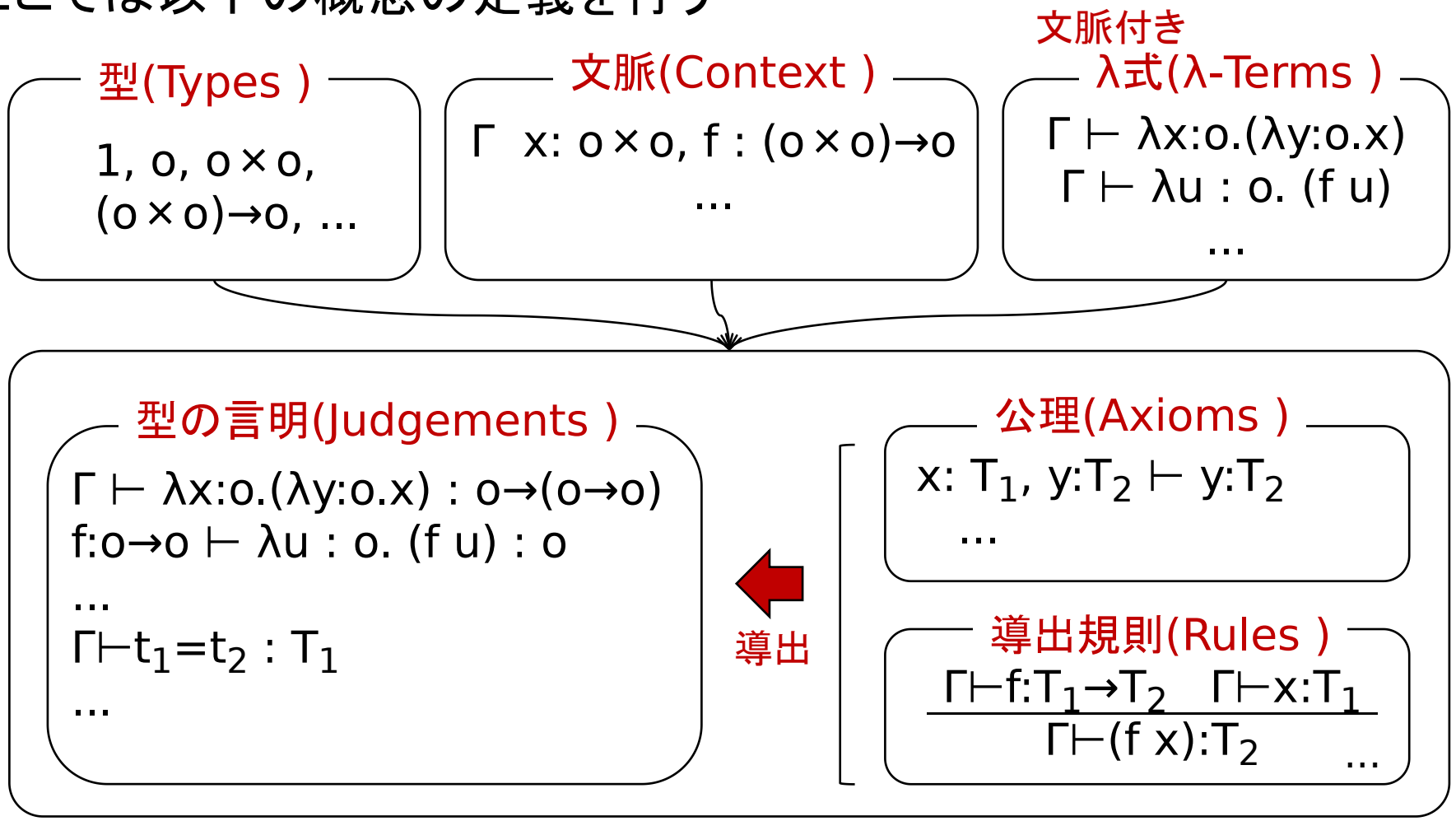
- いろいろな書物では，文脈と単純型付きλ式をそれぞれ定義して，あとで組み合わせる方法をとっている



  - これは，λ式を再帰的に定義するとき，**変数単独**のものをλ式として定義する際に**型が決まっていないという困難**が発生するように思う
  - 「予め型の決まった変数がたくさん用意してある」という設定を置いているものもあるが，λx:T で導入される変数との扱いの違いが気になる
- 本資料ではそれらの書物と異なるが，最初から，**文脈付きのλ式を定義**してみる（直接，上の図の**「型の言明」**を定義）
- 一般の書物で勉強しなおすとき，このことを注意してほしい

単純型付きλ計算(Simply Typed λ-calculus)

- ここでは以下の概念の定義を行う



Simply Typed λ-calculus
型（Types）と文脈（Context）

- **型（Types）**

  基本の型の集合 BasicTypes がある （例 BasicTypes={Bool, Int, Real, ...}）

  - **1** は型である（デカルト閉圏 **C** で終対象 $\mathbf{1}_{\mathbf{C}}$ に対応する特殊な型）
  - BasicTypes の要素は型である
  - α, β が型なら
    - α→β は型である（αからβへの関数の型を意図）
    - α×β は型である（αとβの直積の型を意図）

  補足） 実用の言語でなく理論的な扱いのためだけの場合 BasicTypes はただ一つの型からなる {o} とすることも多い（o はチャーチが命題の型に使ったもの）

- **文脈（Context）**

  変数と型の組の列

  $$x_1:T_1,\ \ldots,\ x_n:T_n$$

  を**文脈（Context）**と言う．0 個の列も許す（ここでは ∅ と書くことにする）

  文脈は **Γ**, **Δ** など，ギリシャ文字の大文字で表す

Simply Typed λ-calculus
文脈付きのλ式（Terms）

- **前提**
  - 変数として使える記号が可算無限個用意されているとする
  - 必要に応じて，型が指定された**定数**（**関数**も含む）を用意してもよい

    c：A, f：A→B, …
- **文脈付きの型付きλ式 (λ-Terms)**
  - **定数**： **c** が型τの定数なら任意の文脈Γに対して，**Γ⊢c:τ** は型付きλ式である
  - **変数**： Γが変数 x に対して，x:τ を含むなら，**Γ⊢x:τ** はτの型付きλ式である
  - *****：**Γ⊢*:1**
  - **組み込みの関数の適用**：
    - **Γ⊢<a, b>:τ×ρ** は型付きλ式．ただし Γ⊢a:τ，Γ⊢b:ρ とする
    - **Γ⊢fst(a) :τ** は型付きλ式．ただし，Γ⊢a:τ×ρ とする．
    - **Γ⊢snd(a):ρ** は型付きλ式．ただし，Γ⊢a:τ×ρ とする．
  - **関数適用**： **Γ⊢(c a):ρ** は型付きλ式．ただし，Γ⊢c:τ→ρ，Γ⊢a:τ とする．
  - **関数抽象化**： **Γ⊢λx:τ.b:τ→ρ** は型付きλ式．ただし，Γ⊢b:ρ とする．

注意）λ式のすべてに型を書くのは，視認性を悪くするので，きちんと型が付くと分かっている場合は，省略することがある（ただし，「λx:T. 」の部分は省略しない）

型の言明(Judgements)

- 言明としては次の2つの形式を扱う
  - **Γ ⊢t : Type**
    - これは**すでに文脈付き単純型付きλ式として定義**した
    - 文脈 Γ の自由変数宣言のもとではλ式 t の型は Type になるという言明である
    - 本資料では，λ式の構成方法として，正しく型が付くλ式しか出来ないようにしたので，これに関する公理も推論ルールも特に設けない
  - **Γ ⊢t=t' : Type**
    - Γ ⊢t:Type　かつ　Γ ⊢t':Type であるとき，**t と t' が等しい**ことを表す言明である
    - 例えば，

      d:o ⊢ (λx:o.x d)=d:o
    - 他に次のようなものもある

      p:o×o ⊢ <fst(p), snd(p)>=p:o×o
    - 単純型付きλ計算には，**理論を構築する必要に応じて等式による公理を入れる**ことで，いろいろな等式が証明できるようになる

単純型付きλ計算の公理

- **公理(スキーム)**
  - Γ ⊢ **fst(<a, b>) = a:τ**
  - Γ ⊢ **snd(<a, b>) = b:σ**
  - Γ ⊢ **<fst(c), snd(c)>=c:τ×σ**
  - Γ ⊢ **((λx:τ.b:σ) a:τ)=b[x:=a] : σ**
    - ここで代入 t[x:=r] は以下のように再帰的に定義されるものとする

x[x:=r] = r	<s, t>[x:=r] = <s[x:=r], t[x:=r]>
y[x:=r] = y	fst(s)[x := r] = fst(s[x:=r])
c[x:=r] = c	snd(s)[x:=r] 0 snd(s[x:=r])
*[x:=r] = *	(s t)[x:=r] = (s[x:=r] t[x:=r])
	(λx.s)[x:=s] = λx.s
	(λy.s)[x:=r] = λy.(s[x:=r]) if x≠y かつ r の自由変数がs[x:=r]で束縛されない

  - Γ ⊢ **λx:τ.(c x)=c:σ**　ただし, Γ ⊢ c:σ かつ x が c に現れない)

(これはη変換の公理. η変換は単純型付きλ計算の公理に含めないこともある)

Simply Typed λ-calculus

- **導出規則 (Rules)**
  - **型の導出**
    - 文脈付き単純λ式の構成方法の中に取り込んだので，特に必要ない
  - **s=t:τ の導出規則**
    - **= の同値性**

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash t=t : \tau} \qquad \frac{\Gamma \vdash s=t : \tau}{\Gamma \vdash t=s : \tau} \qquad \frac{\Gamma \vdash s=t : \tau \quad \Gamma \vdash r=r : \tau}{\Gamma \vdash s=r : \tau}$$

    - **弱化（Weakening）**

$$\frac{\Gamma \vdash s=t : \tau}{\Gamma, x:\sigma \vdash s=t : \tau}$$

Simply Typed λ-calculus

- s=t:τ　続き
  - **product types**

$$\frac{\Gamma \vdash s=u : \tau \qquad \Gamma \vdash t=v : \sigma}{\Gamma \vdash \langle s, t\rangle = \langle u, v\rangle : \tau \times \sigma}$$

$$\frac{\Gamma \vdash s=t:\tau \times \sigma}{\Gamma \vdash fst(s)=fst(t) : \tau}$$

$$\frac{\Gamma \vdash s=t:\tau \times \sigma}{\Gamma \vdash snd(s)=snd(t) : \sigma}$$

$$\frac{\Gamma \vdash s : \tau \times \sigma}{\Gamma \vdash \langle fst(s), snd(s)\rangle = s : \tau \times \sigma}$$

$$\frac{\Gamma \vdash s : \tau}{\Gamma \vdash fst(\langle s, t\rangle)=s : \tau}$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash snd(\langle s, t\rangle)=t : \sigma}$$

Simply Typed λ-calculus

- s=t:τ　続き
  - **function types**

$$\frac{\Gamma \vdash s=t : \tau\to\sigma \qquad \Gamma \vdash u=v : \tau}{\Gamma \vdash (s\ u)=(t\ v) : \sigma}$$

$$\frac{\Gamma, x:\tau \vdash s=t : \sigma}{\Gamma \vdash \lambda x:\tau.s = \lambda x:\tau.t : \tau\to\sigma}$$

$$\frac{\Gamma \vdash s : \tau \quad \Gamma, x:\tau \vdash t : \sigma}{\Gamma \vdash (\lambda x:\tau.t)\ s = t[x:=s] : \sigma}$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \lambda x:\tau.(t\ x) = t : \sigma}$$ （η変換）　ただし，t に x は自由変数として現れないとする

  - **unit type**

$$\frac{\Gamma \vdash t:1}{\Gamma \vdash t=* : 1}$$　unit value * は unit type 1 の唯一の値

単純型付きλ計算で表現できる関数

- 型付きλ計算は再帰関数を定義することができないので，プログラミング言語でいえば，**文の連接と if 文だけでできるプログラム**に相当する
- このことから，ほとんど表現力がないと思うかもしれないが，**λ式での数の表現（チャーチ数）がかなり強力**なので，**足し算**，**掛け算**，**べき乗**など**いろいろな関数を表現**することができる
  - plus := λm.λn.λf.λx.(m f (n f x))
  - mult := λm.λn.λf.(m (n f))　… f を n 回適用することを m 回繰り返す
  - power := λm.λn.(n m)

先に紹介したページのインタープリタのログ

（このインタープリタでは，$n は n のチャーチ数の略記．またλは省略）

```
lb> mult ($2) ($3)
Input = m.n.f.(m (n f)) f.x.(f (f x)) f.x.(f (f (f x)))
      ・・・
Result = f.x.(f (f (f (f (f (f x))))))
f.x. の後ろに f が 2*3=6個ついています.
```

単純型付きλ計算の理論としての表現力
代数などの表現

- 単純型付きλ計算の体系には，BasicTypes，定数，公理（等式）の集合を入れることにより，いろいろな理論を表現することができる
  - **群**
    - **BaiscTypes**={G}
    - **定数**
      - e:G
      - inv:G→G
      - mult:G×G→G
    - **等式**
      - x:G ⊢ mult(x, e)=x:G
      - x:G ⊢ mult(e, x)=x:G
      - x:G ⊢ mult(x, inv(x))=e:G
      - x:G ⊢ mult(inv(x), x)=e:G
      - x:G, y:G, z:G ⊢mult(mult(x, y), z)=mult(x, mult(y, z)):G
  - 同様に**等式で表現することができる代数（普遍代数）**は表現することができる（環や加群など．しかし，体は 1/0 が定義できないため不可）

単純型付きλ計算の理論としての表現力
自己参照的な型の表現

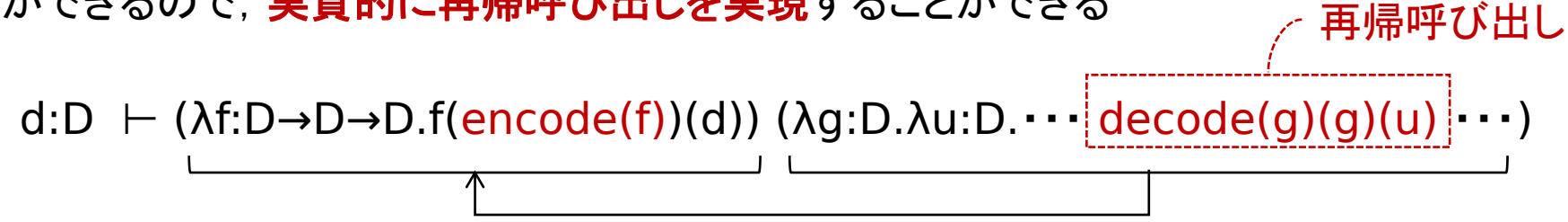
- **自己参照的な型**

  次のような型，定数，等式を加えることにより，関数をデータとして埋め込むことができるデータ型を表現することができる

  - **BasicTypes**={D}
  - **定数**
    - **decode : D→(D→D)**
    - **encode : (D→D)→D**
  - **等式**
    - f:D→D ⊢ **decode(encode(f))=f** : D→D

      つまり，関数 f : D→D を encode(f) : D というデータに変換し，それを decode(encode(f)) とすることで，もう一度もとの関数 f に戻すことができる

  この機構を使えば，自分自身を encode して引数として受け取り，decode して呼び出すことができるので，**実質的に再帰呼び出しを実現**することができる

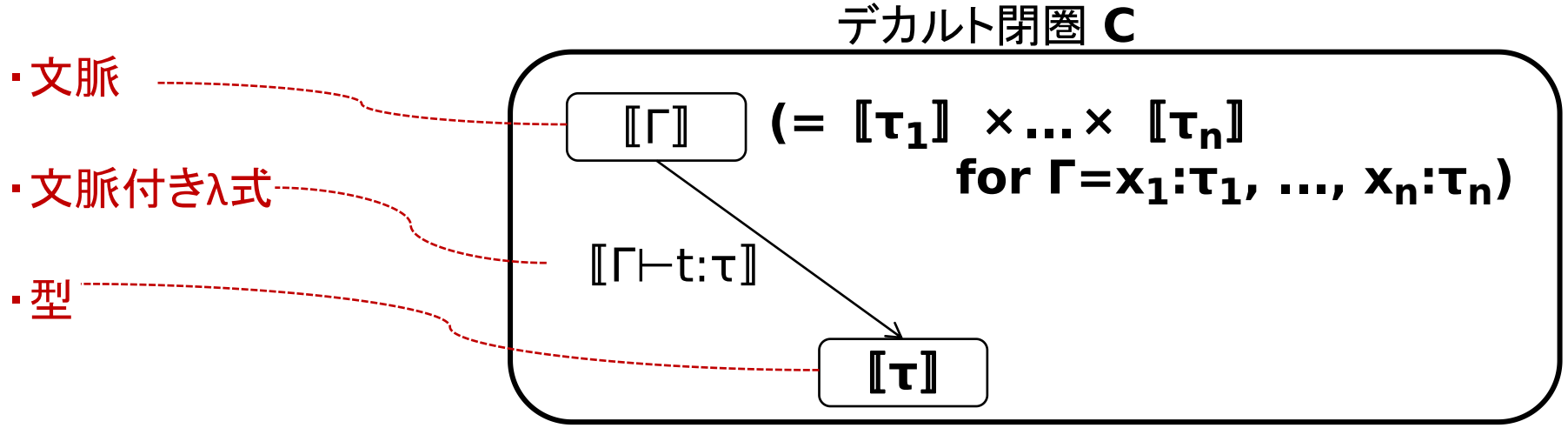


単純型付きλ計算
と
CCC の関係

型付λ計算と CCC の対応

- ここでは，単純型付λ計算にデカルト閉圏（CCC）**C** で意味をつける方法を説明する．意味づけにあたっては次のような対応をとる

型付λ計算	圏（CCC）
型（type）	オブジェクト
文脈（context）	オブジェクト（積オブジェクト）
文脈付きλ式項（λ-term）	射



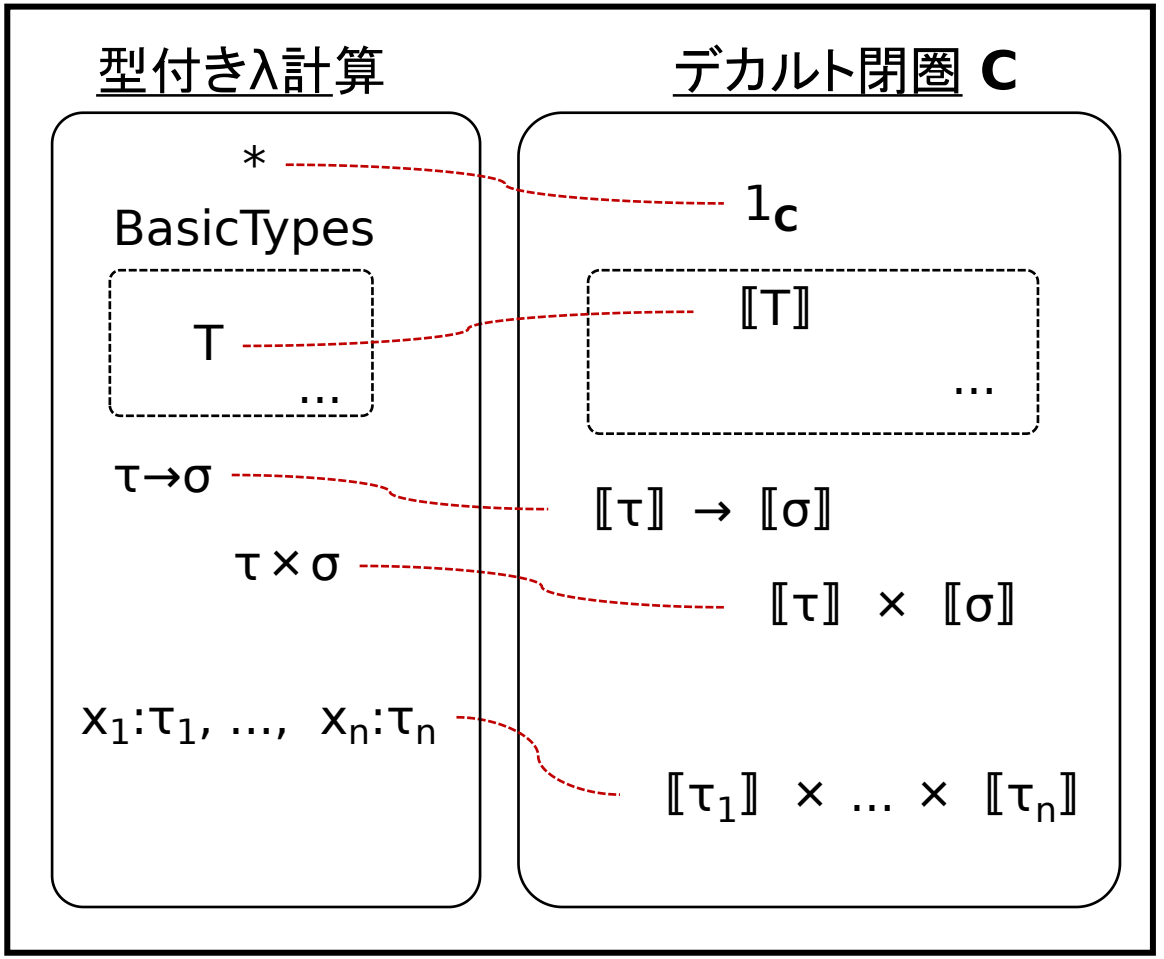
⟦XXX⟧ は XXX を C の中で意味づけしたものを表すとする

単純型付きλ計算の CCC による解釈

- **解釈のルール**

オブジェクトとなるもの

- **型（Type）**
  - 基本型 T には予め **C** の オブジェクト ⟦T⟧ が 割り当てられているとする
  - ⟦τ→σ⟧ = ⟦τ⟧ → ⟦σ⟧
  - ⟦τ×σ⟧ = ⟦τ⟧ × ⟦σ⟧
  - ⟦*⟧ = $1_{\mathbf{C}}$
- **文脈（Context）**
  - ⟦$x_1:\tau_1, \ldots, x_n:\tau_n$⟧ = ⟦$\tau_1$⟧ × … × ⟦$\tau_n$⟧



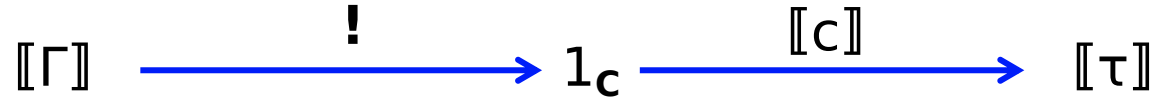
単純型付きλ計算の CCC による解釈

射になるもの

- 文脈付きλ式（Context ⊢ Term）

  ⟦Γ⊢t : τ⟧ は次のように射 ⟦Γ⟧ → ⟦τ⟧ に割り当てられる

  - **定数 ⟦Γ ⊢ c:τ⟧ = ⟦c⟧**

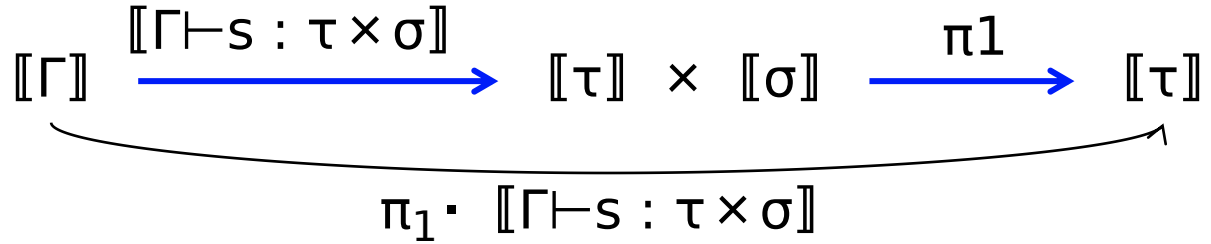
    

    予め定数 c に割り当てられている射

  - **変数 ⟦x1:τ1, …, xn:τn ⊢ xi:τi⟧ =πi**

  - **fst ⟦Γ⊢(fst s) : τ⟧ =π1· ⟦Γ⊢s : τ×σ⟧**

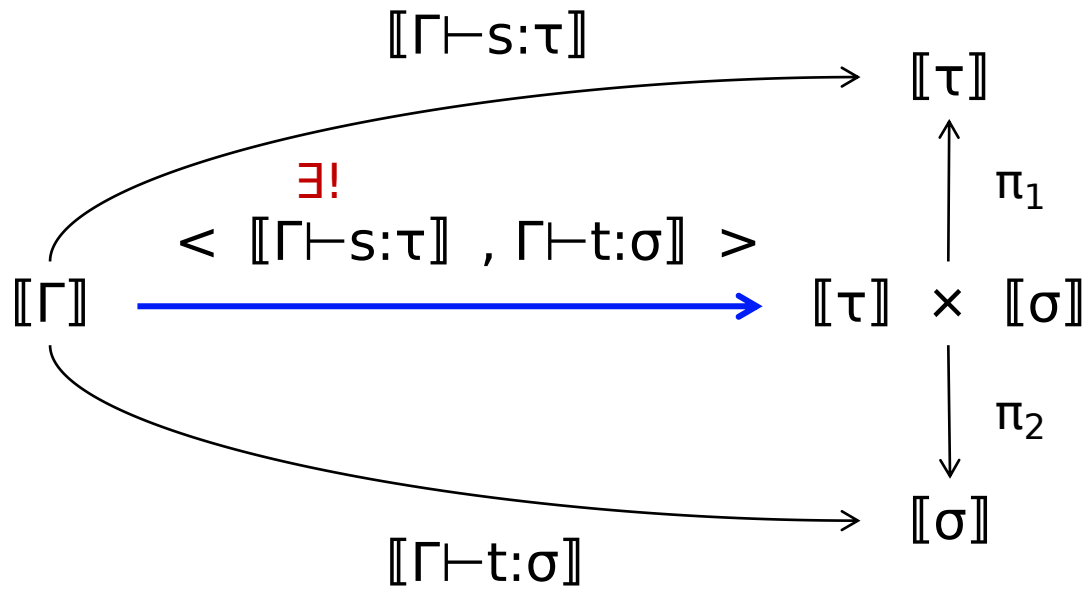
    

  - **snd ⟦Γ⊢(snd s) : σ⟧ =π2· ⟦Γ⊢s : τ×σ⟧**

    上と同様

単純型付きλ計算の CCC による解釈

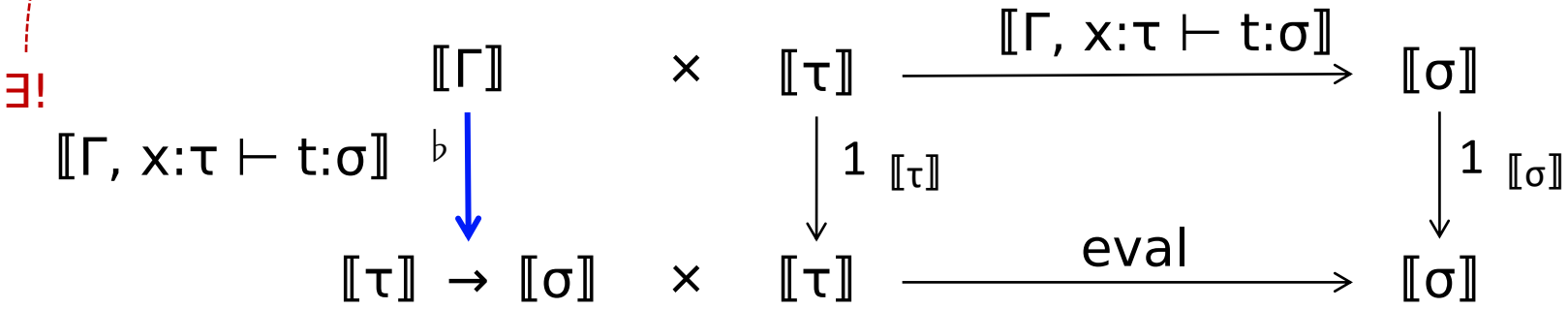
- **ペアを作る関数 ⟦Γ⊢<s, t>:τ×σ⟧ =< ⟦Γ⊢s:τ⟧ , Γ⊢t:σ⟧ >**



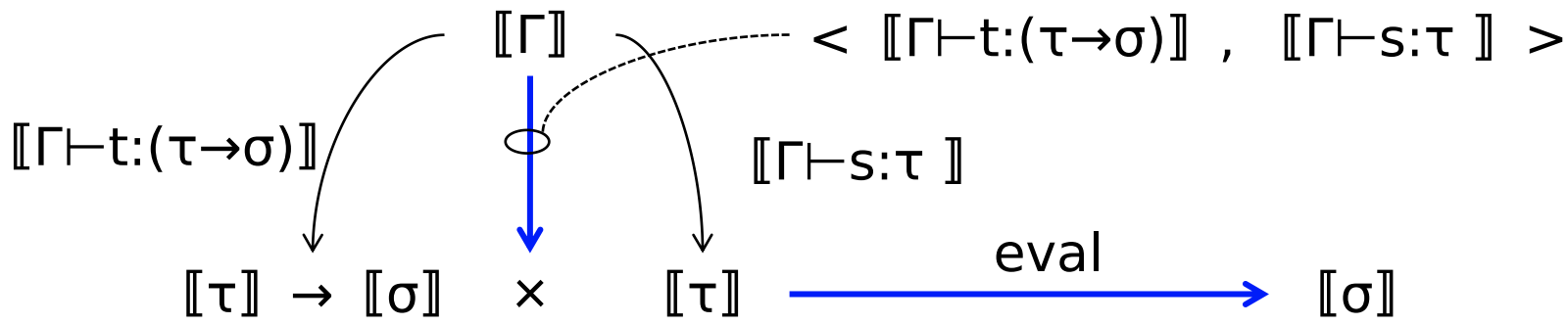
注意）積オブジェクトを，前頁に合わせて横向きに書いていることに注意

[重要] 単純型付きλ計算の CCC による解釈

- **λ抽象化 ⟦Γ⊢λx:τ.t : (τ→σ)⟧ = ⟦Γ, x:τ ⊢ t:σ⟧ ♭**
  - 文脈 Γに宣言 x:τ を加えて，t:σ が導かれていたなら，下の図式のような ⟦Γ⟧ →(⟦τ⟧ → ⟦σ⟧)が一意に存在する

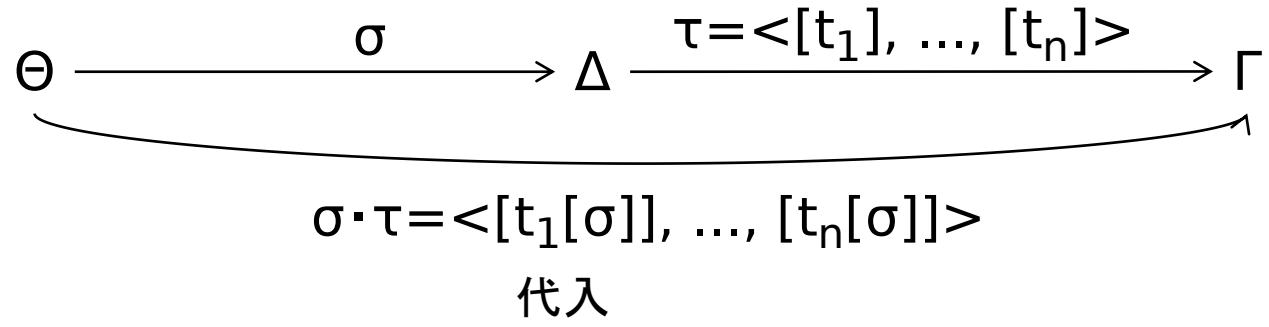


- **λ式の関数適用 ⟦Γ⊢(t s):σ⟧ = eval･< ⟦Γ⊢t:(τ→σ)⟧ , ⟦Γ⊢s:τ ⟧ >**



単純型付きλ計算の項モデル

- 単純型付きλ計算 T から**項モデル**と呼ばれるデカルト閉圏 **C(T)** を次のようにして構成できる
  - **C**(T) のオブジェクト
    - すべての文脈
  - **C**(T) の射
    - $\Delta \rightarrow \Gamma = x_1:T_1, \ldots, x_n:T_n$ は $<[t_1], \ldots, [t_n]>$
      - ここで $\Delta \vdash t_i:T_i$ for $i = 1, \ldots, n$
      - $[t_i]$ は $\Delta \vdash t_i = t'_i : Ti$ が証明できる項を1つにまとめた同値類とする
    - 射の合成



単純型付きλ計算の項モデル

- 項モデル C(T) はデカルト閉圏になる
  - **Γ×Δ** は二つの文脈を並べたもの Γ, Δ
  - **Γ→Δ** は $\Gamma=x_1:T_1, \ldots, x_n:T_n$, $\Delta=y_1:S_1, \ldots, y_m:S_m$ としたとき
    - $\Gamma\to\Delta=f_1:S_1^{\Gamma}, \ldots, f_m:S_m^{\Gamma}$
      ここで $f_i:S_i^{\Gamma}=T_1\to\ldots\to T_n\to S_i$
    - eval : $(\Gamma\to\Delta)\times\Gamma\to\Delta$ は
      $\mathrm{eval} = \langle f_1(x_1)\ldots(x_n), \ldots, f_m(x_1)\ldots(x_n)\rangle$
  - **終対象**は空の文脈 ∅
- C(T) では
  - $\Gamma\vdash t=t':T \Leftrightarrow [\Gamma\vdash t]=[\Gamma\vdash t']$
- したがって，単純型付きλ計算については次の完全性定理が成り立つ

- **定理 （完全性定理）**
  $\Gamma \vdash t=t':T \Leftrightarrow [\Gamma\vdash t] = [\Gamma\vdash t']$ がすべてのデカルト閉圏のモデルで成り立つ

計算機屋さんにとっての成果

- 以上でデカルト閉圏が単純型付きλ計算のモデルになることが分かったが，この事実は計算機屋さんにとって何か良いことがあるのだろうか？
- あまり，「こんなに**素晴らしいこと**がある」と言うものは思いつかないが，なんとなく**良いと思うことを上げていってみよう**
  - 一応，**意味を定めることができる理論的な領域（デカルト閉圏）を得た**ので**心の平安**が得られた
  - 同じことだが，**プログラミング言語の意味を原理的には CCC で定める**ことができるようになった
  - λ式について **t ≠s** は，一つでもこのような CCC があれば良いのだから**楽に示せる場合はある**
  - すべての CCC モデルで統合が成立すれば，単純型付きλ計算でも統合が成立するので**等式を証明する道具は1つ増えた**
  - CCC は **D→D を D に埋め込めるような領域の作**成に役にたつ．．．

まあ，自分で色々やってみたり，論文をサーベイしていけば色々な良いことが見えてくるのかもしれない．

Excuse!

- 以上でデカルト閉圏が単純型付きλ計算のモデルになる話を終わる
- しかし，実は，ここの話は**「関手 F：GraphSyntax→Set がグラフのモデルを与える」**とか，**「一般に，関手 F：C → D が C のモデルを与える」**という**前振りからは，次の2点でずれてきている**
  - デカルト閉圏は圏であるが，**単純型付きλ計算は圏ととらえていない**のでモデルの付与は関手を使っていない
  - **単純型付きλ計算**側も項モデルやそれより緩い（等しいことが証明できる式すべてを必ずしも同一視しない）モデルにより**圏にすることができるが**，その場合の F：λ計算の圏→デカルト閉圏 では，「λ計算の圏」は，**GraphSyntax のようにλ計算の文法を与えている訳ではない**
    - GraphSyntax と同じノリなら，その圏は，変数，定数，文脈，．．．のオブジェクトで構成されるハズである
- うまく話を進めれば繋がるようにできるのかもしれないが，この資料の最初の版ではここで**時間と力が尽きた**ので，このままにしておく
- 次の版で修正するかもしれない

この資料で学習したことと今後の学習のアドバイス

- 圏論の基礎的な概念（**定義，関手，積，冪，デカルト閉圏**）を学んだ
- この範囲で圏論の**応用を感じるトピックス**として次のものに触れた
  - **代数と余代数**による**プログラミング言語の型**の扱い
    - 代数
      - 自然数や**有限リスト**などを表現することができる
      - **始対象の意味論**で帰納法による定義と証明が可能
    - 余代数
      - **無限リスト**や状態遷移マシンを表現することができる
      - **終対象の意味論**で余帰納法による定義と証明が可能
  - **単純型付きλ計算**のモデルと**デカルト閉圏**の同等性
- これらのことを踏まえて学習することで，どこまで我慢すればどのような応用に触れられるかが分かり，**学習の計画が立てやすく**なると思う
- 今後，独力で圏論を学習するためには，**極限・余極限**，**自然変換**，**随伴関手**を深く理解することが学習成功のカギとなる

さらなる学習に向けて

圏論のいろいろなトピック

- **圏論の基礎部分** ．．． **まずは一冊やってみる**

  通常の圏論の基礎的なテキストに載っているトピックスを上げておく．この資料ではこのなかのごく一部（関手，積，冪）について，概念のイメージを添え，応用に関連させて説明した

  - 関手
  - 極限と余極限
  - 冪オブジェクトとデカルト閉圏
  - 自然変換
  - 米田のレンマ
  - 随伴関手
  - モナド
  - トポス

  途中で挫折しないためには，特に
  ・極限，余極限
  ・自然変換
  ・随伴関手
  を十分すぎるくらいしっかり学習することが重要

- **論文などでの応用のサーベイ**

  多少分からないところがあっても応用的な論文を読んでみるのが良いと思う

- **発展的なトピックス**

  圏に演算を入れたり，高階の圏を考えるなど，圏の色々な拡張がある

  モノイダル圏，高階の圏，豊穣圏，．．．

参考文献
計算機科学向きの圏論のテキスト

- **S. Awodey : Category Theory, 2005~**
  - 計算機科学など，マクレーンの Category Theory for Working Mathematicians が想定しているほどの数学的知識を持たない学習者に対するテキストと著者は言っている．単純型付きλ計算や論理学の例などがたくさん載っており，私は結構，興味深いテキストだと思う
  - 私はあまり数学が（英語も）得意ではないので，何回か，最初から読み直した．数回目にこんなことが書いてあったのかと思ったこともあるので，理解の深度が深まった時点で例題などを読み返しても良いと思う
  - 英語で読むのが味わい深くて良いのではないかと思う
- **M. Barr and C. Wells : Category Theory for Computing Science, 1998, 558ページ**
  - タイトルからして計算機科学の学習者向けのテキストである．すでに廃版になっているが，著者らによってPDF の形で公開されている．分厚いが，計算機科学のトピックスがたくさん載っており，あまりしり込みしないで読むのが良いように思う
  - 「モナド」は古い用語の「triple」を使っているが，歴史的経緯を知るには良いのかもしれない

参考文献
圏論一般

- **Tom Leinster : Basic Category Theory, 2014**
  - 計算機科学に関するテキストではないが，短く，簡単に，かつ，直感的な理解を助ける文言が散りばめられているので，最初にざっと読んで，学習すべき概念を頭に入れてから，他のテキストにあたるのも良いかもしれない
  - 随伴関手（Adjoints）が最初の方に説明されている．著者の考えでは，圏論において随伴関手はとても重要であり，また，それほど難しい概念でもないので，学習の早い時期から，この恩恵にあずかるのが良いのではないかとのこと
  - 少し古い arxiv 版は無料で読める
  - 第3章の “Interlude on sets” は集合論に関する話題なのだが，これは，F.W. Lawvere による集合論を圏論的に記述するという話もかなり含んでいる
- **Emily Riehl : Category Theory in Context, 2014, xvii + 240 pages**
  - 良いテキストだと思う．著者本人は簡単な集合論が分かれば分かるだろうと言っているが，私は少し難しめと思う．
  - これも arxiv 版があり，そちらは無料で読める

参考文献
プログラミング言語のデータ型と代数/余代数

- **B. Jacobs , J. Rutten : A Tutorial on (Co)Algebras and (Co)Induction**, 1997, 38 pages
  - 集合の圏を使って具体的に代数，余代数を利用したデータ型への圏論の応用を紹介している.
  - それらのデータ型での証明方法として，（余）帰納法があることが説明されている
- **B. Pierce : A Taste of Category Theory for Computer Scientists**, 1988, 75 pages
  - 計算機科学者に向けた短い圏論のチュートリアル
  - 3部構成になっていて，第2部で当時の計算機科学での応用例が書かれている. 最初の例が，次に述べる萩野氏のカテゴリカルプログラミングに関する短い紹介になっている
- **萩野達也：カテゴリー理論的関数型プログラミング言語**，1990年1月
  - Dialgebra（di- は，両，二の意）や随伴関手を利用したプログラミング言語の提案. ほとんど何も仮定しないレベルから，それらの圏論の原理を使ってデータ型，データ型の構築子，プログラムの構築を可能にしている

参考文献
(単純型付き)λ計算とCCCについて

- **(単純型付き)λ計算**
  - R. Loader : **Notes on Simply Typed Lambda Calculus**, 1998
  - A. Jung : **A short introducion to the Lambda Calculus**, 2004
  - Wikipedia : **Simply typed lambda calculus**

- **単純型付きλ計算とCCC**
  - S. Awodey : Category Theory, 2005 の**第6章 Exponentials**
  - M. Barr and C. Wells : Category Theory for Computing Science, 1998 の**第6章 Cartesian Closed Categories**
  - J. L. Bell : The Development of Categorical Logic, 2005 の**第4章 Cartesian Closed Categories and the λ-Calculus**
  - **その他，多くのレクチャーノート**
    - 単純型付きλ計算とCCC については，インターネット上で検索をかけてみると，大学の授業のレクチャーノートが PDF で見つかる．それらの著者の公開の意思が明確ではないので，ここで個別に URL は書かないが，調べてみるとコンパクトで分かりやすい教材に出会えることがある