「型の理論」入門

Agenda

1. 型のないラムダ計算1（Youtube PDF）
2. 型のないラムダ計算2 (Youtube PDF)
3. 型付きラムダ計算 (Youtube PDF)
4. 論理的推論1 — 判断と論理式 (Youtube PDF)
5. 論理的推論2 — Natural Deduction (Youtube PDF)
6. Curry-Howard対応1 (Youtube PDF)
8. Dependent Type Theory (Youtube PDF)
7. Curry-Howard対応2 –「型」と「証明」 (Youtube PDF)
10. Caliculus of Inductive Construction
9. Homotopy Type Theory (Youtube PDF)

型のないラムダ計算

関数の表記としてのλ記法

- t = x^2+y としよう。このtを、
  xの関数としてみることを λx.t で表し、
  yの関数としてみることを λy.t で表わそう。
- これは、普通の関数で考えると、
  λx.t は、　$f(x)=x^2+y$
  λy.t は、　$g(y)=x^2+y$
  と表記することに相当する。
- t = x^2+y は、また、xとyの二つの変数の関数とみることが出来る。これを λxy.t と表そう。
  λxy.t は、　$h(x, y)=x^2+y$
  と表記することに相当する。

抽象化としてのλ表記

- λ記法を使うことによって、上記の f, g, h のような関数の具体的な名前を使わなくても、関数のある特徴を抽象的に示すことが出来る。これをλによる抽象化と呼ぶ。
- λによる抽象化の例
  - $\lambda x.(x^2 + ax + b)$: xの関数としてみた $x^2 + ax + b$
  - $\lambda a.(x^2 + ax + b)$: aの関数としてみた $x^2 + ax + b$
  - $\lambda b.(x^2 + ax + b)$: bの関数としてみた $x^2 + ax + b$
  - $\lambda xa.(x^2 + ax + b)$: xとaの関数としてみた $x^2 + ax + b$
  - $\lambda xab.(x^2 + ax + b)$: xとaとbの関数としてみた $x^2 + ax + b$

関数の値の計算

- 関数の値の計算をλ表記で考えよう。
- t = x^2+y を
  xの関数とみた時、x=2の時のtの値は 4+y
  yの関数とみた時、y=1の時のtの値は x^2+1
  x,yの関数とみた時、x=2, y=1の時の値は、5
- λで抽象化されたλx.t, λy.tが、例えば、x=2, y=1の時にとる値は、それぞれ、4+y, x^2+1といった関数を値として返す関数であることに注意しよう。
- λxy.tの時、二つの変数 x, yに値を与えると、この関数は、はじめて具体的な値をとる。

λ式での値の適用

- λ式 tに、ある値 sを適用することを、λ式 tの後ろに、値 sを並べて、t sのように表す。
- 例えば、次のようになる。
  $\lambda x.t\ 2 = \lambda x.(x^2+y)\ 2 = 2^2+y = 4+y$
  $\lambda y.t\ 1 = \lambda y.(x^2+y)\ 1 = x^2+1$
  $\lambda xy.t\ 2\ 1 = \lambda xy.(x^2+y)\ 2\ 1 = 2^2+1 = 5$

変数への値の代入

- 関数 f(x) が、x = aの時にとる値は、f(x)の中に現れる x に、x = aの値を代入することで得ることが出来る。これは、通常、f(a)と表わされる。
- 名前のないλ式 tでの変数 xへの値aの代入による関数の値の計算を、次のように表現する。

  t [x:=a]

- もっとも単純なλ式である、x自体が変数である時、代入は次のようになる。

  x [x:=a] = a

適用と代入

- もう少し、適用と代入の関係を見ておこう。
- λ式 λx.tへの、aの適用 (λx.t) aとは、代入 t[x:= a]を計算することである。

  $(\lambda x.t)\ a = t[x:=a]$
- 先の例での値の適用の直観的な説明は、代入を用いると、次のように考えることが出来る。

  $\lambda x.t\ 2 = \lambda x.(x^2+y)\ 2$

  $= (x^2+y)\ [x:=2] = 2^2+y = 4+y$

  $\lambda y.t\ 1 = \lambda y.(x^2+y)\ 1$

  $= (x^2+y)[y:=1]=x^2+1$

型のないラムダ計算の形式化

λ式の形式的定義

- これまでは関数とその引数、引数に具体的な値が与えられた時の関数の値をベースに、λ式とその値の計算を説明してきたが、ここでは、λ式の形式的な定義を与えよう。（正確なものではなく、簡略化したものであることに留意）

1. 変数x,...はλ式である。
2. tがλ式で、xが変数であるなら、λxによる抽象化λx.t はλ式である。(abstraction)
3. t, sがλ式であるなら、tへのsの適用 t s は、λ式である。(application)

λ式の形式的定義

先の定義と、次の定義は
　<name> を「変数」
　<expression>を「λ式」
と読み替えれば、同じものである。

<expression> := <name> | <function> | <application>
<function> := λ <name>.<expression>
<application> := <expression><expression>

自由変数と束縛変数

- 次のような変数を「自由変数」という
  1. 変数x の自由変数は、xだけである
  2. λx. t の自由変数は、x 以外のtの自由変数である
  3. 適用 t s の自由変数は、t の自由変数と s の自由変数を合わせたものである。
- 自由変数以外の変数を「束縛変数」という。

- 例えば、$\lambda x.\, x$ には自由変数はなく、xが束縛変数である
- $\lambda x.\, yx$ の束縛変数はx で、唯一の自由変数はy である

λ計算の形式的ルール

- 次の三つのルールを利用して、λ式を(基本的には単純なものに)変換することをλ計算という。

1. α-conversion:
   抽象化に用いる変数の名前は、自由に変更出来る。例えば、
   $\lambda x.(x^2+1) \Rightarrow \lambda y.(y^2+1) \Rightarrow \lambda z.(z^2+1)$
2. β-reduction:
   代入による計算ルール　$(\lambda x.t)\ a \Rightarrow t[x:=a]$
3. η-conversion:
   $\lambda x.(f\ x) \Rightarrow f$
   xで抽象化されたf(x)は、fに等しいということ。

β-reductionと代入ルール

- 実際のλ計算で、大きな役割を果たすのはβ-reductionである。その元になっている代入ルールを少し詳しく見ておこう。

1. $x[x := N] \equiv N$
2. $y[x := N] \equiv y$, if $x \neq y$
3. $(M1\ M2)[x := N] \equiv (M1[x := N])\ (M2[x := N])$
4. $(\lambda x.M)[x := N] \equiv \lambda x.M$
5. $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N])$, if $x \neq y$, provided $y \notin FV(N)$
   xとyが異なり、yがNの自由変数でないなら、(λx.M)への x:=Nの代入は、Mへのx:=Nの代入である

ちょっと変わったλ式1－ Ω

- λ式　Ω := *(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

ちょっと変わったλ式1－ Ω

- λ式　Ω ：＝　*(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)　第二項の束縛変数の名前をyに変更
  ＝ (λx.xx) (λy.yy)

ちょっと変わったλ式1－ Ω

- λ式　Ω :=　*(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)
  = (λx.xx) (λy.yy)　適用の代入形　β-reduction
  = xx[x:=λy.yy]

ちょっと変わったλ式1－ Ω

- λ式　Ω :=　*(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)
  = (λx.xx) (λy.yy)
  = xx[x:=λy.yy]　　代入の実行
  = (λy.yy) (λy.yy)

ちょっと変わったλ式1－ Ω

- λ式　Ω :=　*(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)
  = (λx.xx) (λy.yy)
  = xx[x:=λy.yy]
  = (λy.yy) (λy.yy)　束縛変数の名前変更
  = (λx.xx) (λx.xx)

ちょっと変わったλ式1－ Ω

- λ式　Ω :=　*(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)
  = (λx.xx) (λy.yy)
  = xx[x:=λy.yy]
  = (λy.yy) (λy.yy)
  = (λx.xx) (λx.xx)
  = Ω

ちょっと変わったλ式1－ Ω

- λ式　Ω :=　*(λx.xx) (λx.xx)* を考えよう。
  これを計算してみる。

  (λx.xx) (λx.xx)
  = (λx.xx) (λy.yy)
  = xx[x:=λy.yy]
  = (λy.yy) (λy.yy)
  = (λx.xx) (λx.xx)
  = Ω

となって、同じλ式Ωが現れ、計算が終わらない。

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g　Y への gの適用
  = λg.(λx.g (x x)) (λx.g (x x)) g

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g 束縛変数名の変更
  = λf.(λx.f (x x)) (λx.f (x x)) g

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf.(λx.f (x x)) (λx.f (x x)) g　　gの適用
  = (λx.g (x x)) (λx.g (x x))

ちょっと変わったλ式2 ー Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf.(λx.f (x x)) (λx.f (x x)) g
  = (λx.g (x x)) (λx.g (x x))　束縛変数名の変更
  = (λy.g (y y)) (λx.g (x x))

ちょっと変わったλ式2 ー Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf.(λx.f (x x)) (λx.f (x x)) g
  = (λx.g (x x)) (λx.g (x x))
  = (λy.g (y y)) (λx.g (x x))　β-reduction
  = g (y y)[y:=(λx.g (x x))]

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf.(λx.f (x x)) (λx.f (x x)) g
  = (λx.g (x x)) (λx.g (x x))
  = (λy.g (y y)) (λx.g (x x))
  = g (y y)[y:=(λx.g (x x))]　　代入の実行
  = g ((λx.g (x x)) (λx.g (x x)))

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf.(λx.f (x x)) (λx.f (x x)) g
  = (λx.g (x x)) (λx.g (x x))
  = (λy.g (y y)) (λx.g (x x))
  = g (y y)[y:=(λx.g (x x))]
  = g ((λx.g (x x)) (λx.g (x x)))
  = g (Y g)

ちょっと変わったλ式2 ― Y combinator

- Y := λg.(λx.g (x x)) (λx.g (x x))とする。
  このとき、Y gを計算してみよう。

  Y g
  = λg.(λx.g (x x)) (λx.g (x x)) g
  = λf.(λx.f (x x)) (λx.f (x x)) g
  = (λx.g (x x)) (λx.g (x x))
  = (λy.g (y y)) (λx.g (x x))
  = g (y y)[y:=(λx.g (x x))]
  = g ((λx.g (x x)) (λx.g (x x)))
  = g (Y g)

すなわち、Y g = g (Y g)となって、Y gは、gの不動点を与える。

ラムダ計算への型の導入

ラムダ計算への型の導入
型σから型τへの関数は、型 $\sigma \rightarrow \tau$ を持つ。

- 型σから型τへの関数は、型 $\sigma \rightarrow \tau$ を持つ。
- あるλ式 e が型 τ を持つことを、$e : \tau$ と表す。
- アルファ変換、ベータ還元、エータ変換のα, β, ηの変換ルールは同じものとする。

型σから型τへの関数は、
型 $\sigma \to \tau$ を持つ。

あるλ式 e が型 τ を持つことを、
$e : \tau$ と表す。

型付きラムダ式の型の定義

型付きλ式の型の定義を次のように行う。

1. 単純な変数 v_i は型を持つ。 $v_i : \tau$
2. $x : \sigma$ で $e : \tau$ なら、$(\lambda x_\sigma.e) : (\sigma \rightarrow \tau)$
3. $e_1 : (\sigma \rightarrow \tau)$で、$e_2 : \sigma$ なら、$(e_1\ e_2) : \tau$

単純な変数 v_i は型を持つ。

v_i ： T

「抽象化」は型を持つ

$x : \sigma$ で $e : \tau$ なら、

$(\lambda x_{\sigma}.\ e) : (\ \sigma \rightarrow \tau\ )$

「適用」は型を持つ

$e_1 : (\sigma \to \tau)$で、$e_2 : \sigma$ なら、

$(e_1\ e_2) : \tau$

この例は、次のように考えるとわかりやすい

「適用」は型を持つ

$e_1 : (\sigma \rightarrow \tau)$で、$e_2 : \sigma$ なら、

$(e_1\ e_2) : \tau$

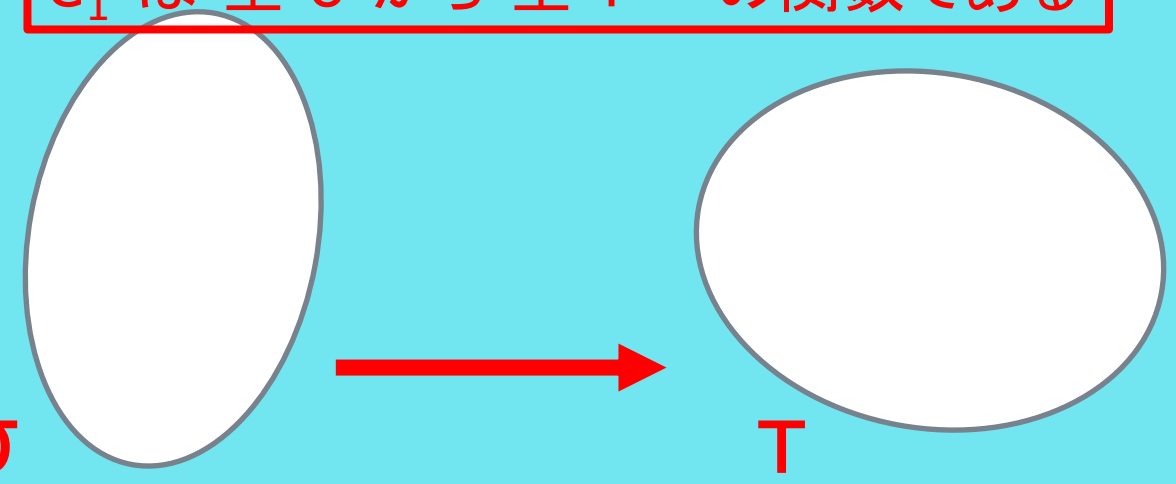
この例は、次のように考えるとわかりやすい

「適用」は型を持つ

$e_1 : (\sigma \rightarrow \tau)$で、

e_1 は 型 σ から 型 τ への関数である

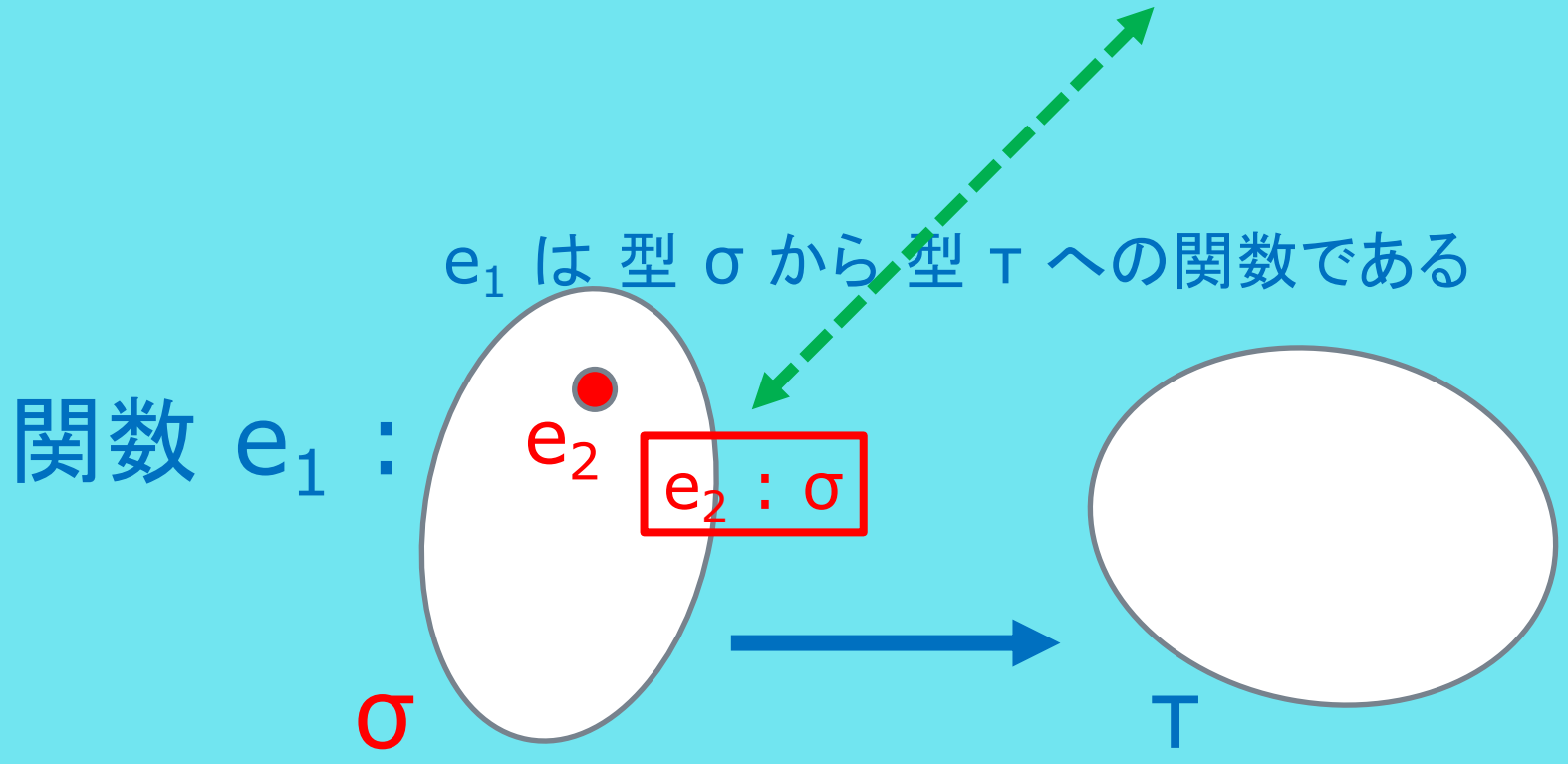
関数 e_1 :



この例は、次のように考えるとわかりやすい

「適用」は型を持つ

$e_1 : (\sigma \rightarrow \tau)$で、$e_2 : \sigma$ なら、

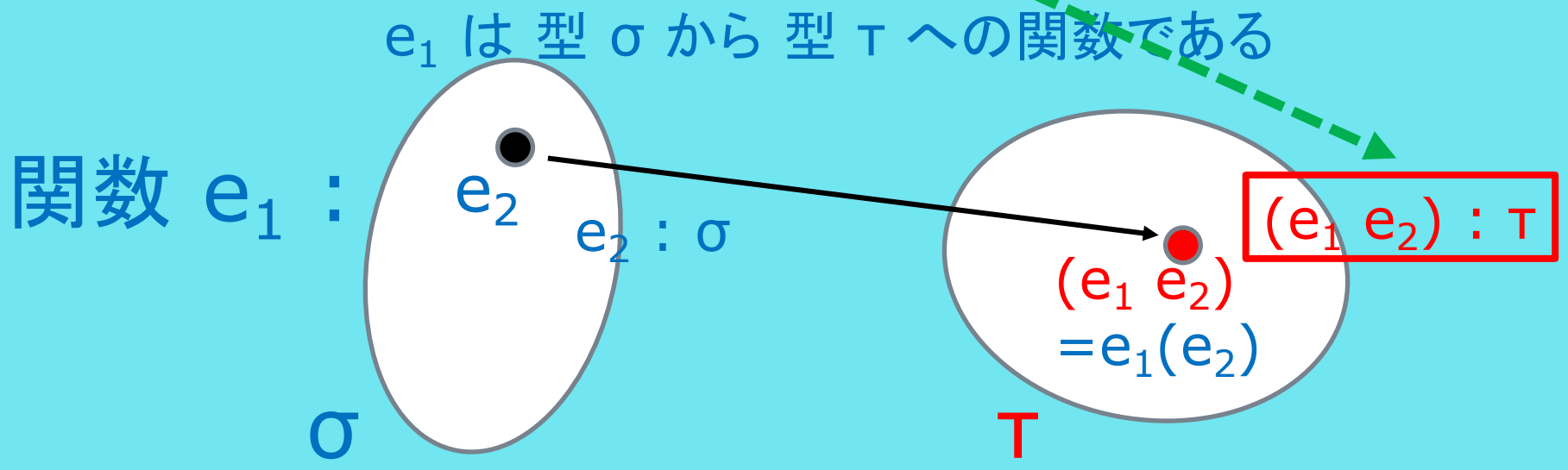


この例は、次のように考えるとわかりやすい

「適用」は型を持つ

$e_1 : (\sigma \rightarrow \tau)$で、$e_2 : \sigma$ なら、

$(e_1\ e_2) : \tau$



型のないラムダ計算と型を持つラムダ計算の違い

- 先に見た型のないλ計算で成り立つ性質の大部分は、型を持つλ計算でも成り立つ。
- ただ、両者のあいだには違いもある。
  型のないλ計算では、任意のλ式に対して任意のλ式の適用を許していたが、型を持つλ計算では、λ式の適用に型による制限が入っている。
- 型を持つラムダ計算では、同じ型を持つλ式同士の $S_\sigma T_\sigma$ という適用は許されない。
- 許されるのは、$S_{\sigma\to\tau}\, T_\sigma$ という型を持つλ式どうしの適用のみである。
- 先の $\Omega := (\lambda x.xx)\ (\lambda x.xx)$ は、同じ型を持つλ式同士の適用なので型を持つラムダ計算では許されないλ式である。

「適用」は型で制限される。可能なのは、

S：(σ → τ)で、T：σ の場合で、
(S T)：τ の場合のみ。

「適用」は型で制限される。可能なのは、

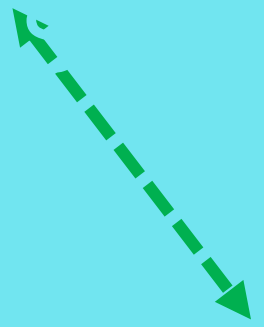
S：(σ → τ)で、T：σ の場合で、
(S T)：τ の場合のみ。

これは、次のようにも表現できる

$S_{\sigma\to\tau} T_{\sigma} : \tau$

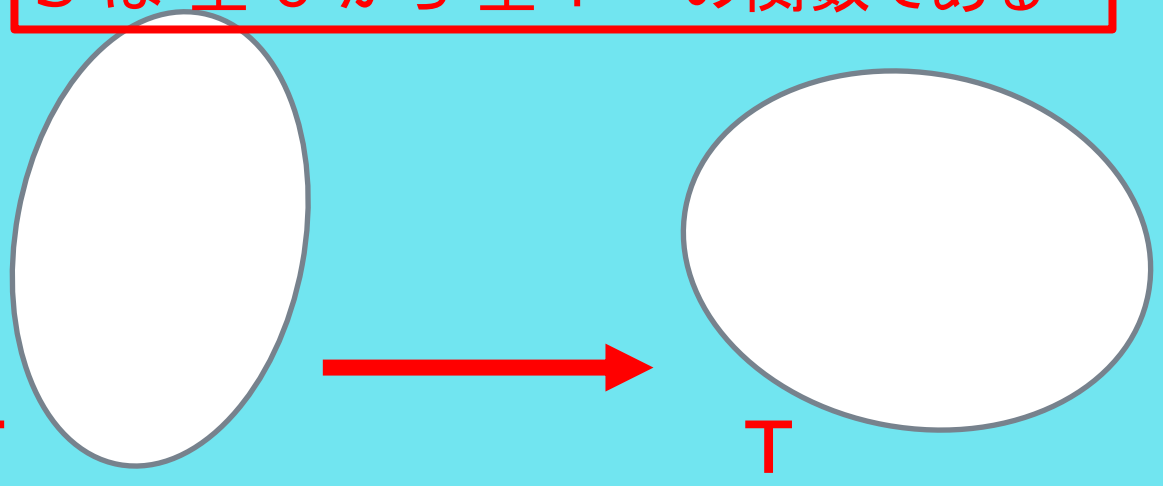
「適用」は型で制限される。可能なのは、

S：(σ → τ)で、



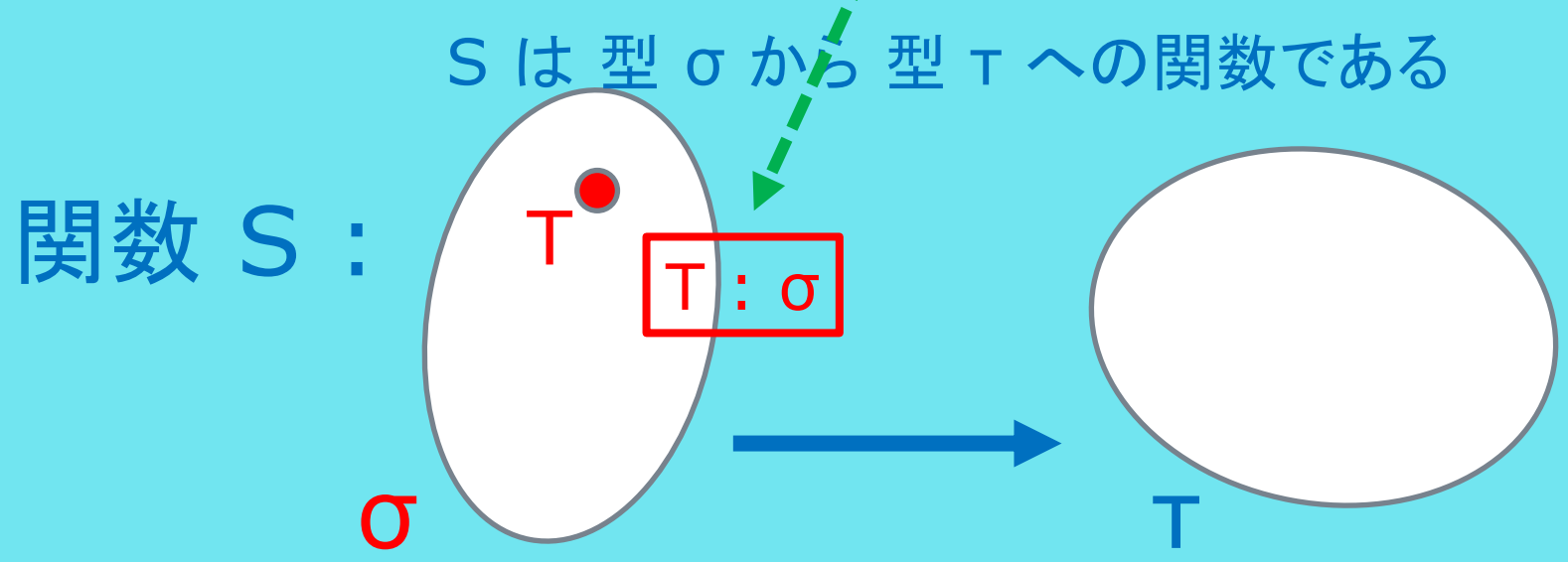
S は 型 σ から 型 τ への関数である

関数 S：



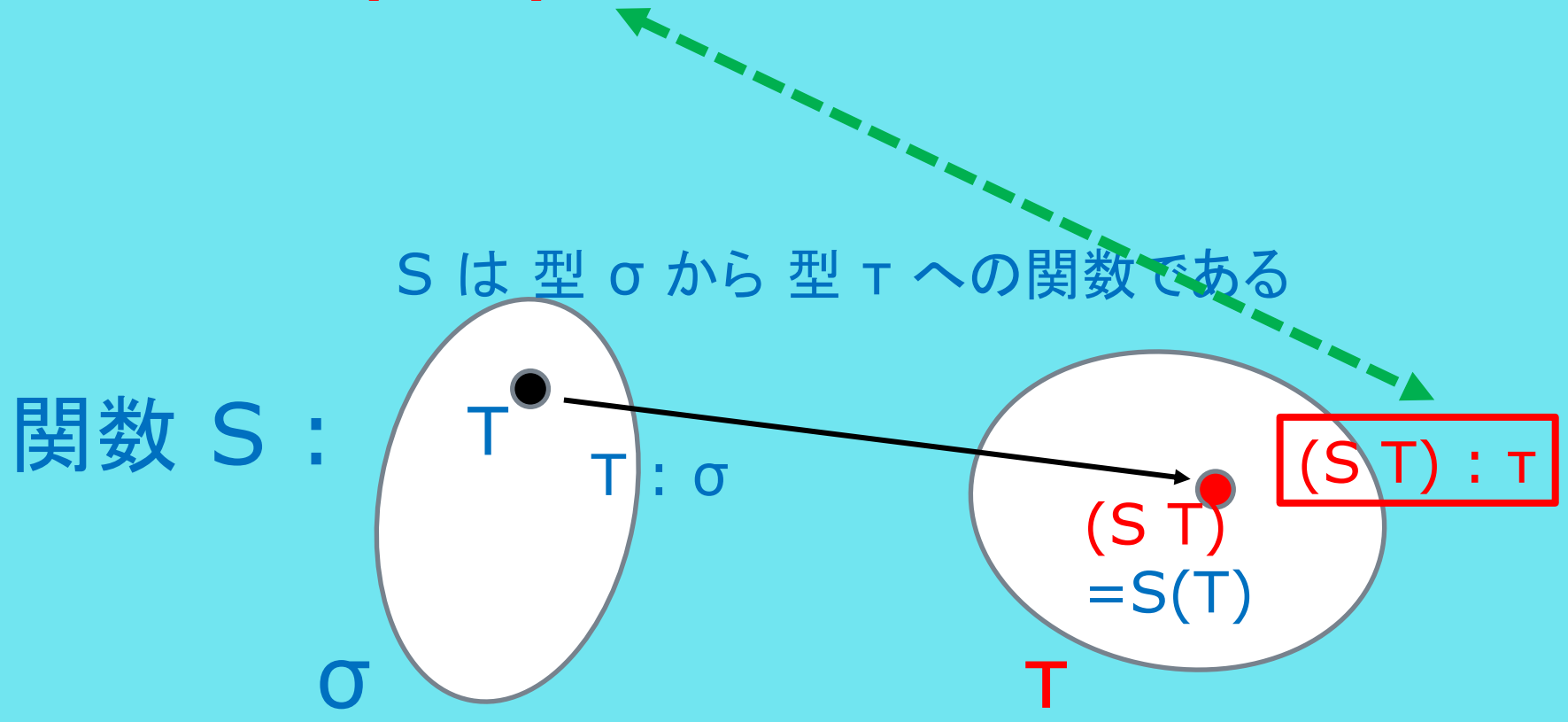
「適用」は型で制限される。可能なのは、

S：(σ → τ)で、T：σ の場合で、
(S T)：τ の場合のみ。



「適用」は型で制限される。可能なのは、

S：(σ → τ)で、T：σ の場合で、
(S T)：τ の場合のみ。



単純な型を持つラムダ計算の特徴

- 単純な型を持つラムダ計算は、こうした点では、型を持たないラムダ計算より表現力が弱いにもかかわらず、次のような重要な特徴を持つ。

- 単純な型を持つラムダ計算では、変換による計算は、必ず停止する。

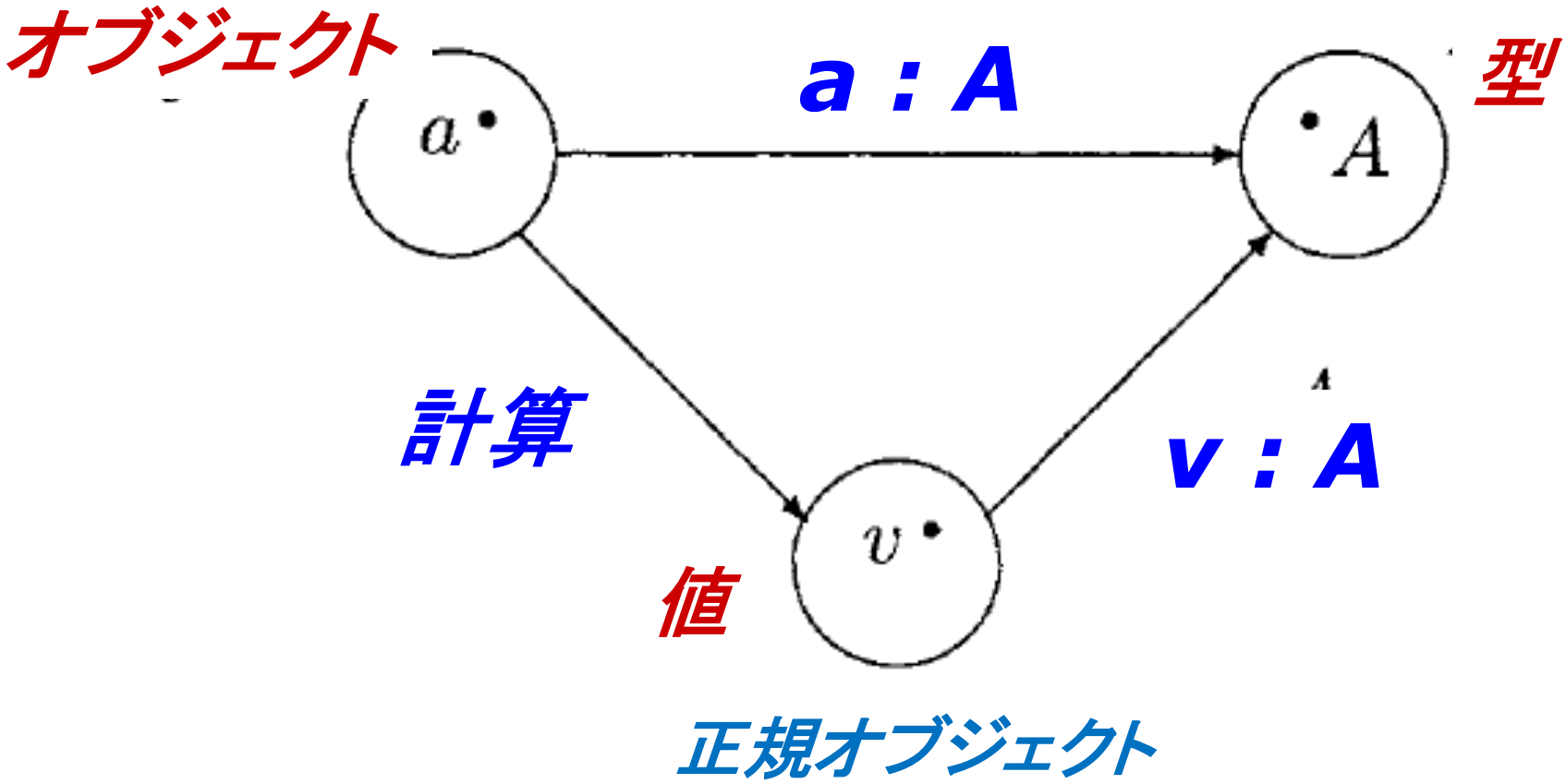
型を持つラムダ計算での計算

- あるオブジェクト aがある型 Aを持つという判断を、 ***a : A*** と表わそう。
- 型を持つラムダ計算の理論では、計算は、次の形をしている。
- 関数 ***λ x:A. b[x]*** を、型Aに属するオブジェクトaを適用して ***b[a]***を得る。
- 計算の下で、全てのオブジェクトはユニークな値を持つ。また、計算で等しいオブジェクトは、同一の値を持つ。

型を持つラムダ計算での
正規オブジェクトとその値

- ある型に属するオブジェクトは、型の計算ルールによって、値が求まるなら、正規オブジェクトと呼ばれる。
- 例えば、１ ＋ １ は、自然数の型に属するが正規ではない。２ は、正規のオブジェクトである。
- ある型 A の正規オブジェクト v は、それ以上計算ルールの適用がが出来ず、それ自身を値として持つ。 この時、***v : A*** と書く。

型、オブジェクト、値



判断と論理式

論理式の意味

- 論理式の意味というと、次のようなことを思い浮かべるかもしれない。

論理式の意味？

論理式	論理式の意味
$A \wedge B$	AかつB
$A \vee B$	AまたはB
$A \rightarrow B$	AならばB
$\sim A$	Aではない
$\forall x P(x)$	全てのxについてP(x)が成り立つ
$\exists x P(x)$	あるxが存在してP(x)が成り立つ

論理式の意味？

- 論理式の意味というと、先のようなことを思い浮かべるかもしれない。
- ただ、それは間違いではないが、不十分である。
- 例えば、ある論理式で表される「命題Aが真である」ということの意味を考えてみよう。

「Aは真である」の意味

- 「Aは真である」 ということは、
- 「私は「Aは真である」ことを知っている」ということである。
- ただ、その前に、知っていることがある。
- それは、「私は「Aは論理式である」ことを知っている」ということである。

判断と命題

- 「Aは真である」 あるいは「Aは論理式である」といった言明を「判断」と呼ぶ。
- 「判断」の概念は、常に、「命題」の概念に先立つ。
- 「判断」における論理的帰結の概念は、「命題」における含意より先に、説明されなければならない。

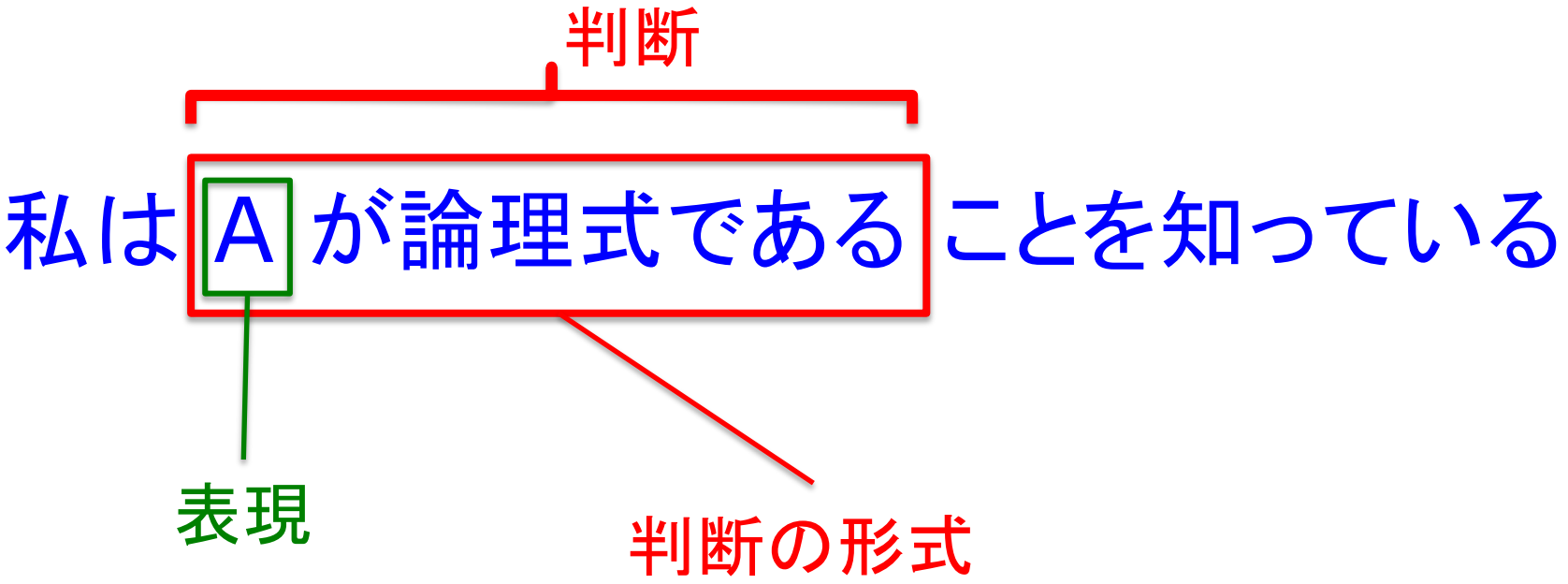
論理式の表現･構成についての判断の構造

私は A が論理式である ことを知っている

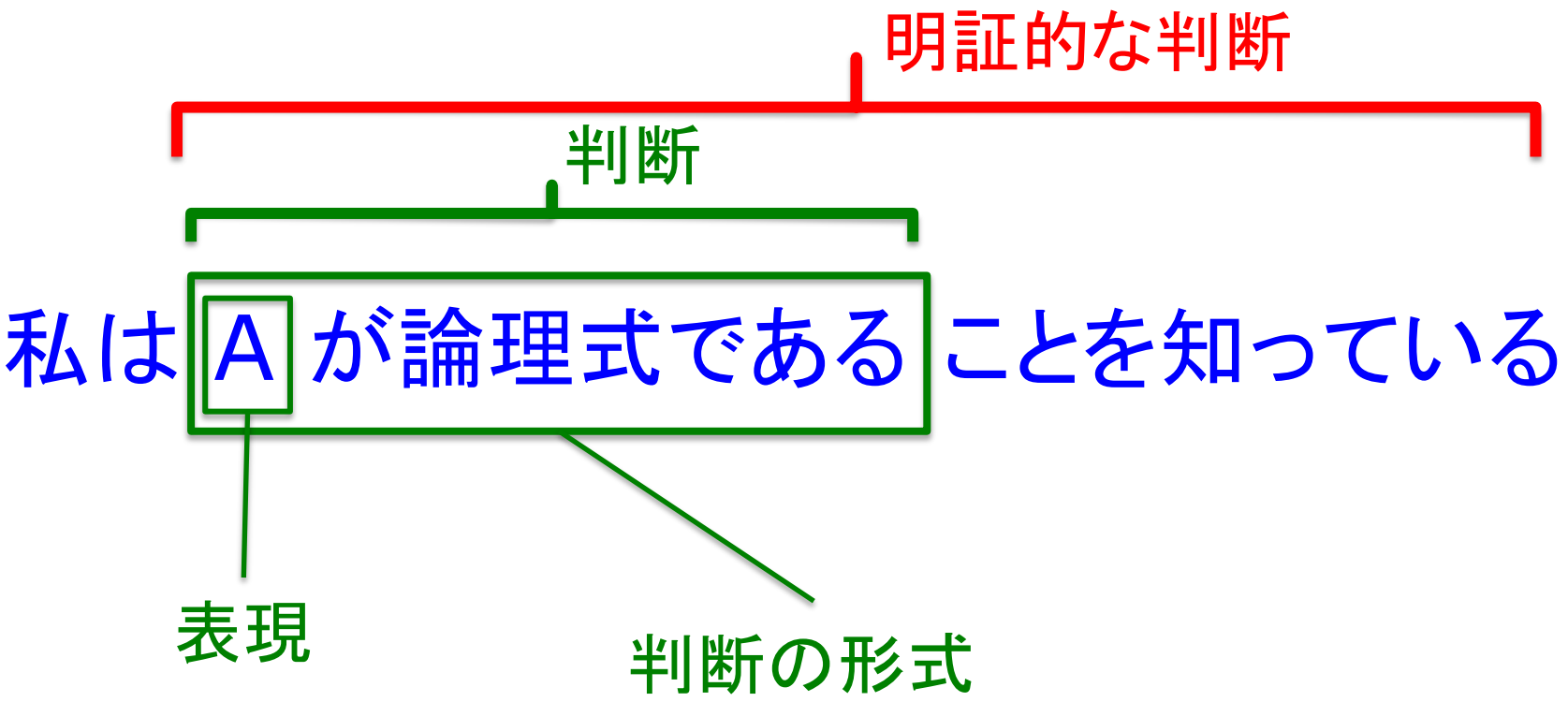
論理式の表現・構成についての判断の構造

私は A が論理式である ことを知っている

論理式の表現･構成についての判断の構造



論理式の表現･構成についての判断の構造



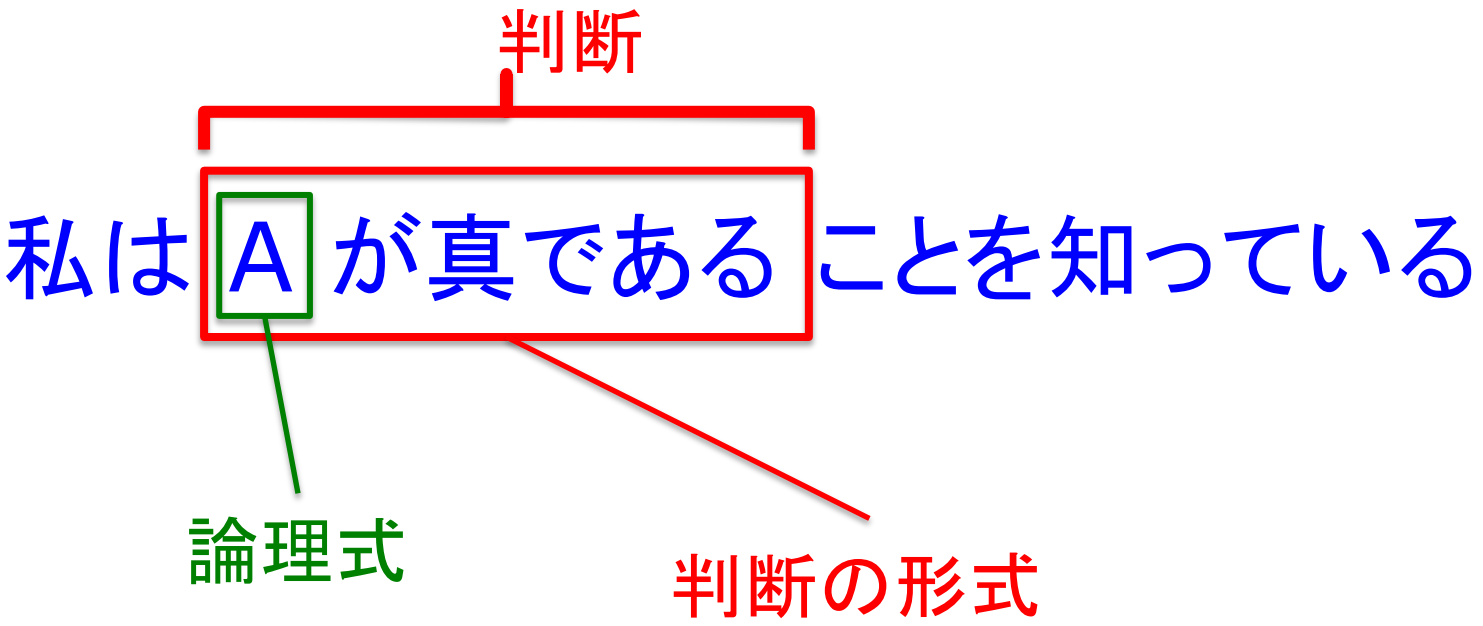
論理式の正しさについての判断の構造

私は A が真である ことを知っている

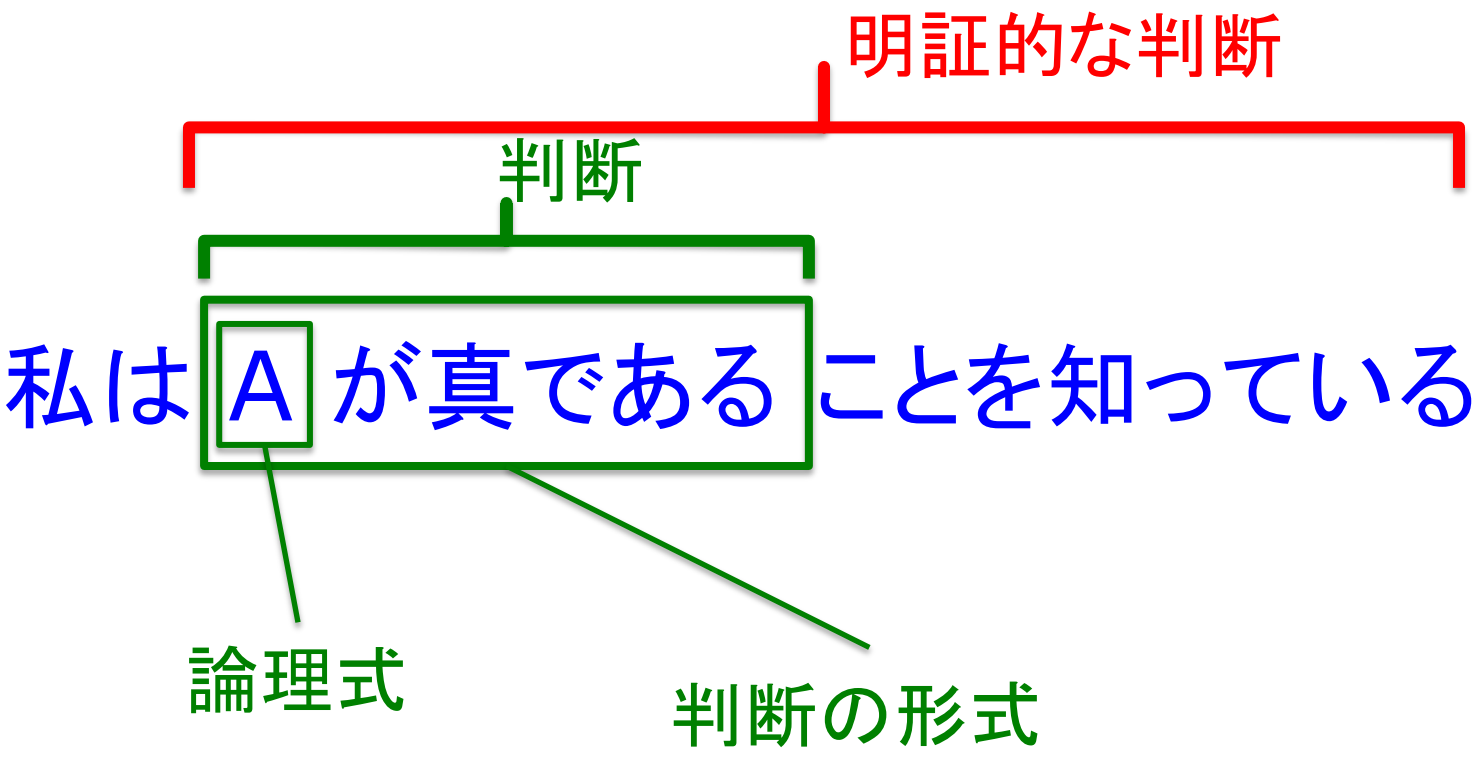
論理式の正しさについての判断の構造

私は A が真である ことを知っている

論理式の正しさについての判断の構造



論理式の正しさについての判断の構造



証明とは何か？

- 証明とは、判断を明証なものにするもの。
- 証明すること＝知ること＝理解して把握すること
- 知るようになる＝知識を得ること
- 証明と知識は、同じもの

判断の論理的帰結の表現

- 論理学者は、横棒が好きである。
- 「判断２」が「判断１」の論理的帰結であることを次のように表す。

$$\frac{\text{判断１}}{\text{判断２}}$$

- こうした表現を用いて、論理式の構成ルールや、論理式の推論ルールを表現することができる。

論理式の構成ルール (Formation Rule)

- $\dfrac{x \text{ は変数である} \quad P \text{ は命題である}}{(\forall x.\ P) \text{ は命題である}}$ (∀F)

- $\dfrac{x \text{ は変数である} \quad P \text{ は命題である}}{(\exists x.\ P) \text{ は命題である}}$ (∃F)

- $\dfrac{}{\bot \text{は命題である}}$ (⊥F)

(⊥は、矛盾を表す)

推論ルール
Natural Deduction

Natural Deduction（命題論理版）

- ここでは、Gentzen によって導入され、Prawitzらによって整理された「自然な演繹」の推論ルールを与える Natural Reductionのシステムを紹介する。
- 今回紹介するのは、(∀x. P) 、(∃x. P) といった 量化子 Quantifier を含まない、その「命題論理」への制限版である。

∧を含む推論ルール

- 「命題Aは真である」という「判断」を、⊢ Aと表すことにしよう。
- 「命題Aが真」で「命題Bが真」という判断があるなら、「命題 (A ∧ B)は真」であるという判断が成り立つ。
- この推論ルールを次のように表す。

$$\frac{\vdash A \qquad \vdash B}{\vdash (A \wedge B)}$$

- もしも、上段、下段に登場しているのが「～は真である」という判断であることが明確であれば、次のように記号⊢ を省略しても構わない。

$$\frac{A \qquad B}{(A \wedge B)}$$

∧を含む推論ルール

- $\dfrac{A \qquad B}{(A \wedge B)}$

  という推論ルールでは、上段にはなかった論理記号 ∧ が下段には登場する。
- この推論ルールを「∧ 導入ルール」と呼び、次のように横棒の隣に **∧I** と書く。(∧ **I**ntroduction の意である。)

$$\frac{A \qquad B}{(A \wedge B)}\ \wedge\mathbf{I}$$

∧を含む推論ルール

- 「(A∧B) は真」だとしよう。この時、「Aは真」も「Bは真」も成り立つ。このルールは次のように表現される。

$$\frac{(A \wedge B)}{A} \qquad \frac{(A \wedge B)}{B}$$

- この推論ルールでは、上段にあった論理記号∧は、下段では削除されている。
- この推論ルールを「 ∧ 削除ルール」と呼び、次のように横棒の隣に **∧E** と書く。(**∧ E**limination の意である。)

$$\frac{(A \wedge B)}{A}\ \wedge E_1 \qquad \frac{(A \wedge B)}{B}\ \wedge E_2$$

- 添字の1,2は、上段の第一項、あるいは、第二項が下段に残ることを表している。

vを含む推論ルール

□「Aが真」であれば、「(A ∨ B) は真」である。また
「Bが真」であれば、やはり、「(A ∨ B) は真」である。

$$\frac{A}{(A \vee B)}\ \vee I_1 \qquad \frac{B}{(A \vee B)}\ \vee I_2$$

vを含む推論ルール

- 「Aが真」であれば、「(A ∨ B) は真」である。また「Bが真」であれば、やはり、「(A ∨ B) は真」である。

$$\frac{A}{(A \vee B)}\ \vee\mathbf{I}_1 \qquad \frac{B}{(A \vee B)}\ \vee\mathbf{I}_2$$

- この推論ルールでは、上段にはなかった論理記号 ∨ が下段には導入される。
- この推論ルールを「 ∨ 導入ルール」と呼び、上のように横棒の隣に ∨**I** と書く。(∨ **I**ntroduction の意である。)
- 添字の1,2は、下段の第一項、あるいは、第二項が上段にあることを表している。
- vを含む推論ルールにも、「∨ 削除ルール」があるのだが、その説明は、ここでは省略する。

→ を含む推論ルール

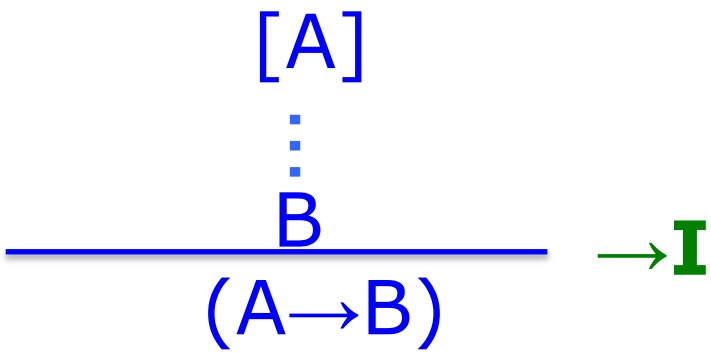
- 「(A → B) が真」で「Aが真」なら、「Bは真」である。

$$\frac{(A \rightarrow B) \qquad A}{B} \rightarrow E$$

- このルールは、代表的な推論ルールである「三段論法」を表している。Modus Ponens あるいは Cutとも呼ぶことがある。
- この推論ルールでは、上段にあった論理記号→は、下段では削除されているので、削除ルールである。

→ を含む推論ルール

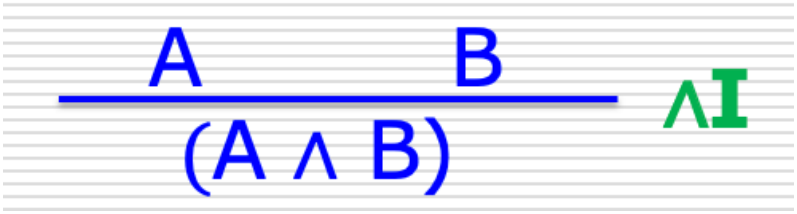
- 推論の途中で A, Bが導かれるなら、「(A → B) は真」である。
- ここで [A] は、Aの導出の経過が省略されていることを表している。途中の ⋮ も同様に導出の省略を表す。

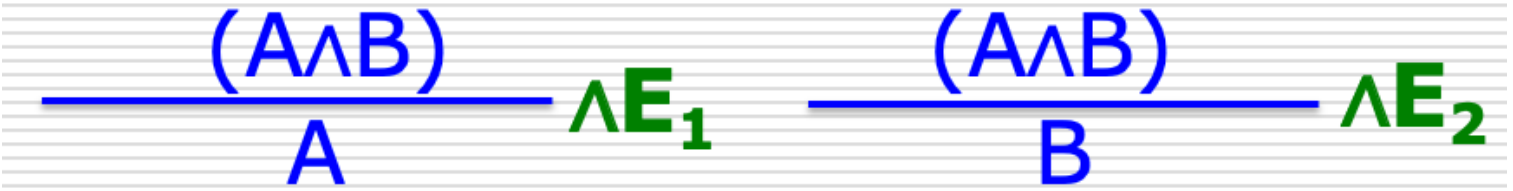


- この推論ルールでは、上段にはなかった論理記号 ∨ が下段には導入されているので、「 導入ルール」である。

Natural Deduction（命題論理版）の推論ルールのまとめ

- ∧を含む推論ルール

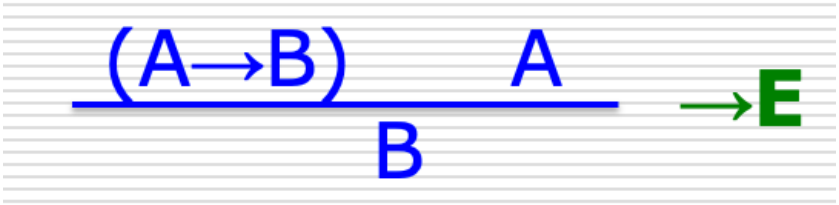


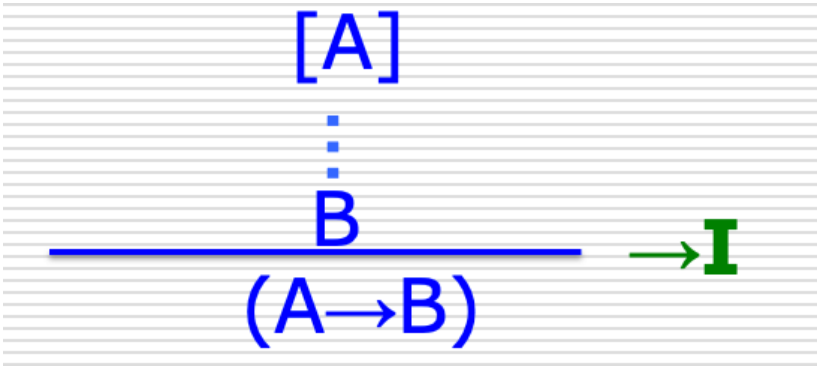


- ∨を含む推論ルール

Natural Deduction（命題論理版）の推論ルールのまとめ

- → を含む推論ルール





命題 $A \to (B \to (A \wedge B))$ の導出例

命題 $A \to (B \to (A \wedge B))$ の導出例

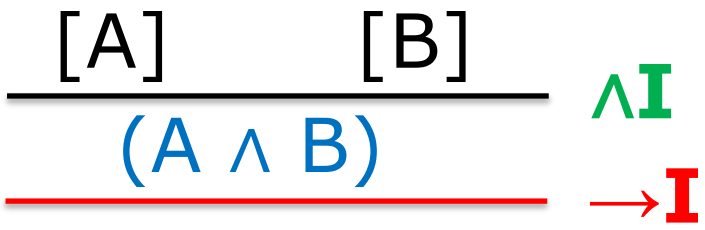
[A]　　[B]

命題 A → (B → (A ∧ B)) の導出例

命題 $A \to (B \to (A \wedge B))$ の導出例

$$\dfrac{[A] \qquad [B]}{(A \wedge B)}\ \wedge\mathbf{I}$$

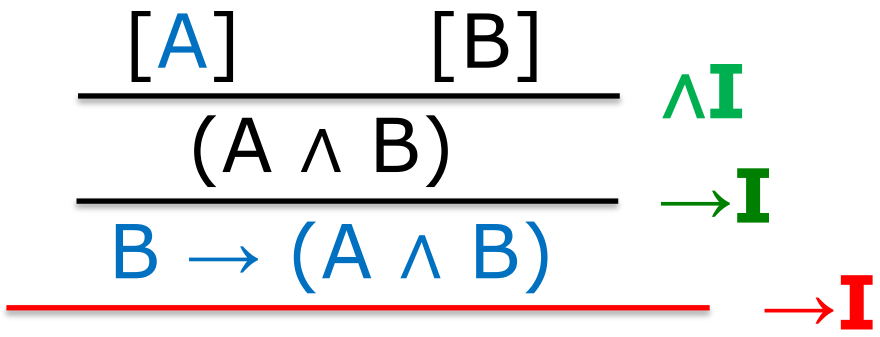
命題 $A \to (B \to (A \wedge B))$ の導出例



命題 $A \to (B \to (A \wedge B))$ の導出例

$$\dfrac{\dfrac{[A] \qquad [B]}{(A \wedge B)}\,\wedge\mathbf{I}}{B \to (A \wedge B)}\,\to\mathbf{I}$$

命題 $A \to (B \to (A \wedge B))$ の導出例



命題 $A \to (B \to (A \wedge B))$ の導出例

$$\dfrac{\dfrac{\dfrac{[A] \qquad [B]}{(A \wedge B)}\,\wedge\mathbf{I}}{B \to (A \wedge B)}\,\to\mathbf{I}}{A \to (B \to (A \wedge B))}\,\to\mathbf{I}$$

命題 $A \to (B \to (A \wedge B))$ の導出例

$$\dfrac{\dfrac{\dfrac{[A] \qquad [B]}{(A \wedge B)}\,\wedge\mathbf{I}}{B \to (A \wedge B)}\,\to\mathbf{I}}{A \to (B \to (A \wedge B))}\,\to\mathbf{I}$$

Curry-Howard対応

Curry-Howard対応

- 1940年代のChurchの仕事から、ラムダ計算に対する関心が、ふたたび活発化するのは、20年近くたった1960年代からである。
- 理由ははっきりしている。コンピュータが現実に動き出したからである。
- この時期の代表的な成果は、HowardのCurry-Howard対応の研究である。

Curryの発見
型付きラムダ計算と直観主義論理との対応

- 既に1934年に、Curryは、型付きラムダ計算と直観主義論理とのあいだに、対応関係があることを発見していた。

- 型を持つラムダ計算の関数の型を示す矢印 → と、論理式“A → B” の中に出てくる含意を意味する矢印 → との間に、対応関係があるというのだ。

Curry, Haskell (1934), "Functionality in Combinatory Logic”
http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1076489/pdf/pnas01751-0022.pdf

Howardによる発展
論理式はラムダ計算の型と見なすことが出来る

- Howardは、このCurryの発見を、さらに深く考えた。
- 1969年の彼の論文のタイトルが示すように、論理式は型付きラムダ計算の型と見なすことが出来るのである。
- ただし、彼のこの論文が公開されたのは、1980年になってからのことらしい。

Howard,Williams (1980)
"The formulae-as-types notion of construction”
http://www.cs.cmu.edu/~crary/819-f09/Howard80.pdf

「型」と「命題」、「要素」と「証明」との「双対性」

- Curry-Howard対応の、中心的な概念は、「型」と「命題」、「要素」と「証明」との「双対性」である。

- 「*p* が命題 *P* の証明である」ことを、***p : P***
  と表すことができる。
- 「*p* は、型 *P* の要素である」ことも、***p : P***
  と表すことができる。

「型」と「命題」、「要素」と「証明」との「双対性」

- Curry-Howard対応の、中心的な概念は、「型」と「命題」、「要素」と「証明」との「双対性」である。

- 「*p* が命題 *P* の証明である」ことを、***p : P***
  と表すことができる。
- 「*p* は、型 *P* の要素である」ことも、***p : P***
  と表すことができる。

- 「*p* が命題 *P* の証明である」という判断を表す ***p : P*** は、「*p* は、型 *P* の要素である」という判断を表す ***p : P*** と見なすことが出来るし、また、逆の見方も出来るのである。

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との「双対性」

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との「双対性」

「*p* は、命題 *P* の証明である」

証明　　命題

$$p : P$$

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との「双対性」

$$p : P$$

要素　　型

「p は、型 P の要素である」

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との「双対性」

「*p* は、命題 *P* の証明である」

証明　　命題

$$p : P$$

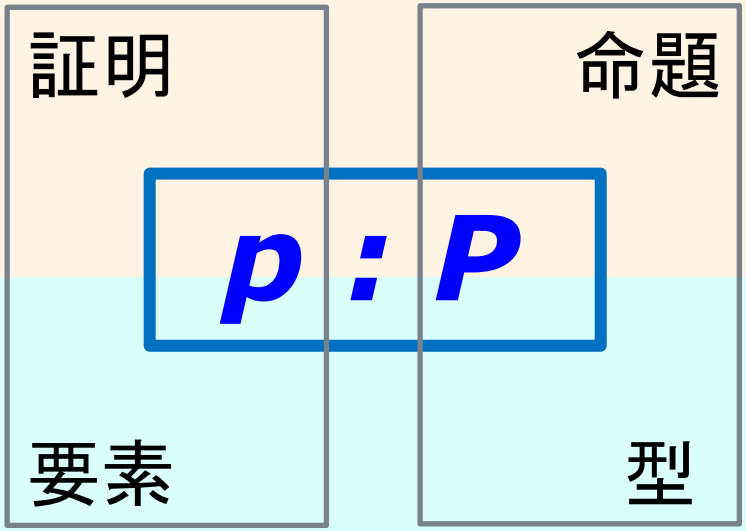
要素　　型

「*p* は、型 *P* の要素である」

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との「双対性」

「*p* は、命題 *P* の証明である」

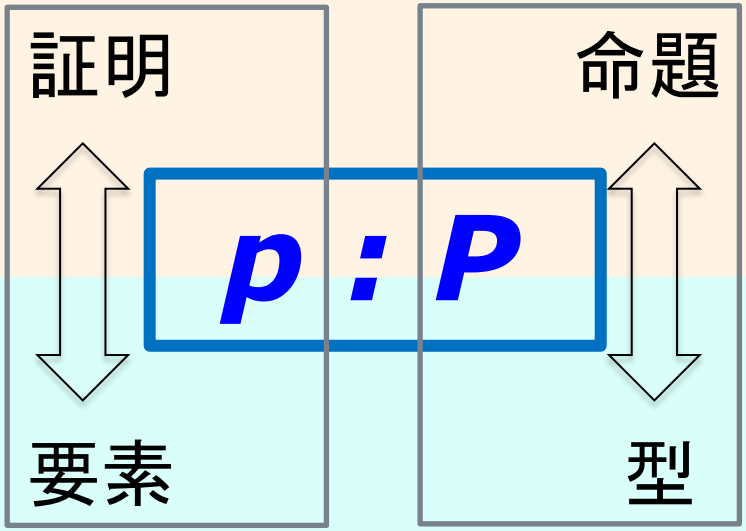


「*p* は、型 *P* の要素である」

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との「双対性」

「p は、命題 P の証明である」

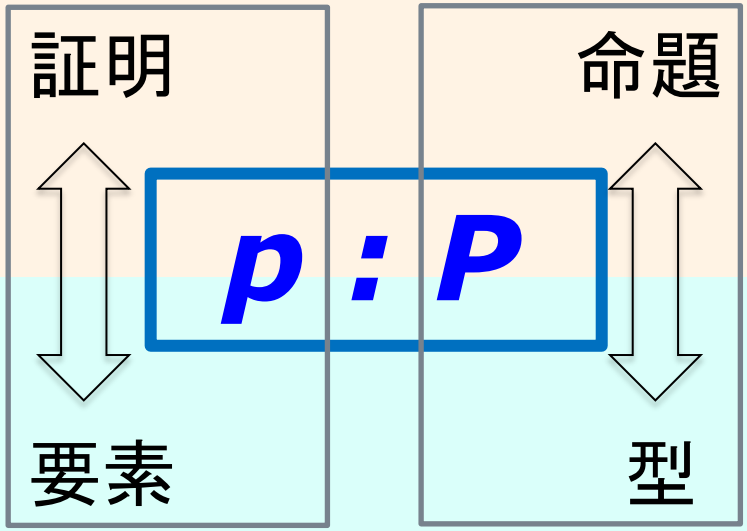


「p は、型 P の要素である」

Curry-Howard対応

「型」と「命題」、「要素」と「証明」との「双対性」

「*p* は、命題 *P* の証明である」



証明は
要素である

命題は
型である

「*p* は、型 *P* の要素である」

“Propositions as Types”
“Proofs as Terms”

- Howardの洞察は、”Propositions as Types”, “Proofs as Terms” (これを、PATという)として、あとにみるMartin-Löfの型の理論に大きな影響を与えた。
- 同時に、Curry-Howard対応は、型付きラムダ計算をプログラムと見なせば、”Proof as Program”としても解釈出来る。
- こうした観点は、今日のCoq等の証明支援言語の理論的基礎になっている。

Dependent Type

Dependent Type Theory

- 現在の「型の理論」の基本が出来上がるのは、1980年代になってからである。その中心人物は、Martin-Löfである。
- 現在のCoq, Agdaという証明支援システムは、彼のDependent Typeの理論に基礎付けられている。

命題への型の拡張

- Churchの単純な型を持つラムダ計算の体系は、基本的には、型A, 型Bに対して、型 A → Bで表現される関数型の型しか持たない。
- それに対して、Martin-Löfは、論理的な命題にも、自然なスタイルで型を定義した。例えば、Aが型であり、Bが型であれば、A∧BもA→BもA∨Bも型であるというように。もちろん、それぞれは異なる型である。
- すでに、先のCurry-Howard対応の証明では、その事実を使ってきた。（前後が逆だった。）
- ある aが型Aを持つ時、***a : A*** で表す。

同一性への型の付与

- Martin-Löf は、式 a = b にも型を与える。
  $a =_A b : Id(A\ a\ b)$
  型 Id(A a b) を持つ式は、a と b は等しいという意味を持つ。
- $a =_A b$ のAは、型Aの中での同一性であることを表している。すなわち、a : A で b : Aで、かつa = bの時にはじめて、$a =_A b$ は型Id(A a b)
  を持つ。
- ここでは式の同一性について述べたが、式の同一性と型の同一性は、異なるものである。

型の理論の記述

- Martin-Löfの型の理論は、次のことを示す、四つの形式で記述される。

1. ある対象aが、型であること
   $a : Type$
2. ある表現式 aが、型 αの要素であること
   $a : \alpha$
3. 二つの表現式が、同じ型の内部で、等しいこと
   $a = b : \alpha$
4. 二つの型が、等しいこと
   $\alpha = \beta : Type$

Dependent Type

- これまで、A∧B, A → B, A∨B といった、元になる型 A, B を指定すると新しい型が決まるといったスタイルで型を導入してきた。
- これとは異なる次のようなスタイル、型Aそのものではなく型Aに属する要素 a：Aに応じて新しい型 B(a)が変化するような型の導入が可能である。
- 例えば、実数R上のn次元のベクトル Vec(R,n)とn+1次元のベクトルVec(R,n+1)は、別の型を持つのだが、これは nに応じて型が変わると考えることが出来る。

Dependent Type

- こうした型をDependent Typeと呼び、次のように表す。
  $\Pi(x:A).B(x)$
- Dependent Typeは、ある型Aの値aにディペンドして変化する型である。
- 先の例のベクトルの型は、次のように表される。
  $\Pi(x:N).Vec(R,n)$
  これは、全てのnについてVec(R,n)を考えることに対応している。
- 型の理論では、全称記号は、Dependent Typeとして導入される。

Dependent TypeとPolymorphism

- Dependent Typeの例を、もう一つあげよう。
  n次元のベクトルは、自然数 N、整数 Z、実数 R、複素数 C上でも定義出来る。
  { N, Z, R, C } : T とする時、次のように定義される Dependent Typeを考えよう。
  $$\Pi(x : T)Vec(x,n)$$
- これは、次元nは同じだが、定義された領域が異なる別の型 Vec(N,n),Vec(Z,n),Vec(R,n), Vec(C,n)を考えることに相当する。
- こうして、Polymorphicな関数は、Dependent Typeとして表現されることが分かる。

Inductive Type

- 型の理論では、基本的な定数と関数から、新しい型を帰納的に定義することが出来る。こうした型をInductive Type（帰納的型）と呼ぶ。
- 次は、そうした帰納的な定義の例である。
  ここでは、自然数の型natが、ゼロを意味する定数Oとsuccessor関数を表す関数Sで、帰納的に定義されている。

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.
```

「型の理論」と「証明の理論」

証明の解釈

- Kolmogorovは、命題 a → bの証明に、「命題 aの証明を命題 bの証明に変換する方法を構築すること」という解釈を与えた。
- このことは、a → bの証明を、aの証明からbの証明への関数と見ることが出来るということを意味する。
- このことは、同時に、命題をその証明と同一視出来ることを示唆する。
- 型はこうした証明の集まりを表現し、そうした証明の一つは、対応する型の、一つの項と見なすことが出来る。

命題の証明は、何から構成されるか？

- $\bot$ なし
- **$A \wedge B$** Aの証明とBの証明の両方
- **$A \vee B$** Aの証明、あるいは、Bの証明
- **$A \rightarrow B$** Aの証明からBの証明を導く方法
- **$(\forall x)B(x)$** 任意のaに対してB(a)の証明を与える方法
- **$(\exists x)B(x)$** あるaに対するB(a)の証明

命題の証明は、何から構成されるか？ 形式的に

⊥	none
A ∧ B	Aの証明であるaと、 Bの証明であるbのペア**(a,b)**
A ∨ B	Aの証明である**i(a)**、あるいは、 Bの証明である**j(b)**
A → B	Aの証明であるaに対して、 Bの証明b(a)を与える**(λx)b(x)**
(∀x)B(x)	任意のaに対して Bの証明b(a)を与える**(λx)b(x)**
(∃x)B(x)	あるaと、B(a)の証明であるbのペア **(a,b)**

Curry-Howard対応を
どう証明するか？

基本的な方針

Curry-Howardの対応関係をどう証明するか？

- ここでは、Simon Thompsonのやり方に従う。
- まず、***p : P*** が「*p* が命題 *P* の証明である」という判断を表すとして、そうした判断が従うべきルールを形式的に記述する。
- ついで、***p : P*** が「*p* が型 *P* の要素である」という判断を表すとして、そうした判断が従うべきルールを形式的に記述する。
- そうすると、前者と後者の形式的記述が、その解釈は別にして、まったく同じものであることが分かる。

命題とその証明のルールとその解釈

- **Formationルール**
  システムの証明は、どのような形をしているか？

- **Introduction / Eliminationルール**
  どのような表現が、命題の証明になるか？

- **Computationルール**
  表現は、どのようにして単純な形に還元されるのか？

型とその要素のルールとその解釈

- **Formationルール**
  システムの型は、どのような形をしているか？

- **Introduction / Eliminationルール**
  どのような表現が、型の要素となるか？

- **Computationルール**
  表現は、どのようにして単純な形に還元されるのか？

Curry-Howard対応の証明

ここでは、

→ と ∨ を含む場合の証明を行う

→ を含む場合の対応の証明

ここでは、

ここでは、先に「命題の証明」のケースを取り上げ、ついで「型の要素」のケースについて取り上げる。

解釈は異なっているが、両者の形式的記述が同一であることをチェックしてほしい。

→ を含む命題の証明のルール

- Aが命題であり、Bが命題であれば、A→Bは命題になる。（**Formationルール**）

→ を含む命題の証明のルール

- Aが命題であり、Bが命題であれば、A→Bは命題になる。
- xが命題Aの証明だと仮定して、eが命題Bの証明だとしよう。この時、命題A→Bの証明は、x:Aであるxの関数としてみた (λx:A).eである。
  （**Introduction**ルール）

→ を含む命題の証明のルール

- Aが命題であり、Bが命題であれば、A→Bは命題になる。
- xが命題Aの証明だと仮定して、eが命題Bの証明だとしよう。この時、命題A→Bの証明は、x:Aであるxの関数としてみた (λx:A).eである。
- qが命題A→Bの証明で、aが命題Aの証明だとすれば、関数qにaを適用した(q a)は、命題Bの証明である。（**Eliminationルール**）

→ を含む命題の証明のルール

- Aが命題であり、Bが命題であれば、A→Bは命題になる。
- xが命題Aの証明だと仮定して、eが命題Bの証明だとしよう。この時、命題A→Bの証明は、x:Aであるxの関数としてみた (λx:A).eである。
- qが命題A→Bの証明で、aが命題Aの証明だとすれば、関数qにaを適用した(q a)は、命題Bの証明である。
- この時、*((λx:A).e) a = e[a/x*（eの中のxを全てaで置き換えたもの）になる。
  （**Computationルール**）

→ を含む命題の証明のルール

- Formationルール

$$\frac{A \text{ は命題である} \qquad B \text{ は命題である}}{(A \to B) \text{ は命題である}} \ (\to \mathbf{F})$$

- Introductionルール

$$\frac{\begin{array}{c}[x : A] \\ \vdots \\ e : B\end{array}}{(\lambda x{:}A).e : (A \to B)} \ (\to \mathbf{I})$$

→ を含む命題の証明のルール

- Eliminationルール

$$\frac{q : (A \to B) \qquad a : A}{(q\ a) : B}\ (\to \mathbf{E})$$

- Computationルール

$$((\lambda x{:}A).e)\ a \to e[a/x]$$

→ を含む要素の型のルール

- Aが型であり、Bが型であれば、A→Bは型になる。
  （**Formationルール**）

→ を含む要素の型のルール

- Aが型であり、Bが型であれば、A→Bは型になる。
- xが型Aの要素だと仮定して、eが型Bの要素だとしよう。この時、型A→Bの要素は、x:Aであるxの関数としてみた、(λx:A).eである。
  （**Introduction**ルール）

→ を含む要素の型のルール

- Aが型であり、Bが型であれば、A→Bは型になる。
- xが型Aの要素だと仮定して、eが型Bの要素だとしよう。この時、型A→Bの要素は、x:Aであるxの関数としてみた、(λx:A).eである。
- qが型A→Bの要素で、aが型Aの要素だとすれば、関数qにaを適用した(q a)は、型Bの要素である。（**Elimination**ルール）

→ を含む要素の型のルール

- Aが型であり、Bが型であれば、A→Bは型になる。
- xが型Aの要素だと仮定して、eが型Bの要素だとしよう。この時、型A→Bの要素は、x:Aであるxの関数としてみた、(λx:A).eである。
- qが型A→Bの要素で、aが型Aの要素だとすれば、関数qにaを適用した(q a)は、型Bの要素である。
- この時、*((λx:A).e) a = e[a/x]*（eの中のxを全てaで置き換えたもの）になる。
  （**Computationルール**）

→ を含む型のルール

- Formationルール

$$\frac{A \text{ は型である} \qquad B \text{ は型である}}{(A \to B) \text{ は型である}} \ (\to \mathbf{F})$$

- Introductionルール

$$\frac{\begin{array}{c}[x : A] \\ \vdots \\ e : B\end{array}}{(\lambda x{:}A).e : (A \to B)} \ (\to \mathbf{I})$$

→ を含む型のルール

- Eliminationルール

$$\frac{q : (A \to B) \qquad a : A}{(q\ a) : B}\ (\to \mathbf{E})$$

- Computationルール

$$((\lambda x{:}A).e)\ a \to e[a/x]$$

V を含む場合の対応の証明

ここでは、

ここでは、ワンステップごとに、「命題の証明」のケースと「型の要素」のケースの場合を交互に取り上げた。

ここでも、解釈は異なっているが、両者の形式的記述が同一となることをチェックしてほしい。

∨ を含む命題の証明のルール

- Aが命題であり、Bが命題であれば、A∨Bは命題になる。
- qが命題Aの証明であれば、qは命題A∨Bの証明を、rが命題Bの証明であれば、rも命題A∨Bの証明を与えるように見える。
- ここで、qに「qは∨で結ばれた命題の左の式の証明である」という情報を付け加えたものを、inl qとし、同様に、rに「rは∨で結ばれた命題の右の式の証明である」という情報を付け加えたものを、inr rと表すことにしよう。

v を含む型の要素のルール

- Aが型であり、Bが型であれば、AvBは型になる。
- qが型Aの要素であれば、qは型AvBの要素を、rが型Bの要素であれば、rも型AvBの要素を与えるように見える。
- ここで、qに「qはvで結ばれた型の左の型の型である」という情報を付け加えたものを、inl qとし、同様に、rに「rはvで結ばれた型の右の型の要素である」という情報を付け加えたものを、inr rと表すことにしよう。

∨ を含む命題の証明のルール

- その上で、qが命題Aの証明であれば、inl qが命題A∨Bの証明を、rがBの証明であれば、inr rが命題A∨Bの証明を与えると考える。
- このことは、命題A∨Bが証明されるのは、命題Aまたは命題Bのどちらかが証明される場合に限ることになる。
- 例えば、命題A∨~Aが証明されるのは、命題Aまたは命題~Aのどちらかが証明された場合だけということになる。この体系では、一般に「排中律」はなりたたない。

∨ を含む型の要素のルール

- その上で、qが型Aの要素であれば、inl qが型A∨Bの要素を、rが型Bの証明であれば、inr rが型A∨Bの要素を与えると考える。
- このことは、型A∨Bが要素を持つのは、型Aまたは型Bのどちらかが要素を持つ場合に限ることになる。
- 例えば、型A∨~Aが要素を持つのは、型Aまたは型~Aのどちらかが要素を持つ場合だけということになる。この体系では、一般に「排中律」はなりたたない。

v を含む命題の証明のルール

- 今、pが命題A∨Bの証明で、fが命題A→Cの証明で、gが命題B→Cの証明とした時、命題Cの証明がどのような形になるか考えよう。
- pが命題A∨Bの命題Aの証明であるか（inl p）
  命題A∨Bの命題Bの証明であるか(inr p)の場合に応じて、命題Cの証明の形は変わる。
- 前者の場合、pは命題Aの証明であるので、
  f:A→Cと組み合わせれば、命題Cが導ける。
  後者の場合、pは命題Bの証明であるので、
  g:B→Cと組み合わせれば、命題Cが導ける。

∨ を含む型の要素のルール

- 今、pが型A∨Bの要素で、fが型A→Cの要素で、gが型B→Cの要素とした時、型Cの要素がどのような形になるか考えよう。
- pが型A∨Bの型Aの要素であるか（inl p）
  型A∨Bの型Bの要素であるか(inr p)の場合に応じて、型Cの要素の形は変わる。
- 前者の場合、pは型Aの要素であるので、f:A→Cと組み合わせれば、型Cが導ける。
  後者の場合、pは型Bの要素であるので、g:B→Cと組み合わせれば、型Cが導ける。

∨ を含む命題の証明のルール

- Formationルール

$$\frac{A \text{ は命題である} \qquad B \text{ は命題である}}{(A \vee B) \text{ は命題である}} \ (\vee F)$$

- Introductionルール

$$\frac{q : A}{inl\ q : (A \vee B)} \ (\vee I_1) \qquad \frac{r : B}{\mathrm{inr}\ r : (A \vee B)} \ (\vee I_2)$$

- Eliminationルール

$$\frac{p : (A \vee B) \quad f : (A \to C) \quad g : (B \to C)}{cases\ p\ f\ g : C} \ (\vee E)$$

∨ を含む型のルール

- Formationルール

$$\frac{A \text{ は型である} \qquad B \text{ は型である}}{(A \vee B) \text{ は型である}} \ (\vee \mathbf{F})$$

- Introductionルール

$$\frac{q : A}{\mathit{inl}\ q : (A \vee B)} \ (\vee \mathbf{I}_1) \qquad \frac{r : B}{\mathrm{inr}\ r : (A \vee B)} \ (\vee \mathbf{I}_2)$$

- Eliminationルール

$$\frac{p : (A \vee B) \quad f : (A \to C) \quad g : (B \to C)}{\mathit{cases}\ p\ f\ g : C} \ (\vee \mathbf{E})$$

∨ を含む命題の証明のルール

- Computationルール

  cases (inl q) g f → f q

  cases (inr r) g f → g r

∨ を含む型のルール

- Computationルール

  cases (inl q) g f → f q

  cases (inr r) g f → g r

v を含む型のルール

- 型AvBは、型Aと型Bの直和である。
- 通常のプログラム言語では、値に名前のタグがつけられたレコード型と考えてよい。
- Eliminationルールは、その名前による場合分けに相当する。
- Computationルールは、pにAのタグがついていれば、f p を計算してCの要素を求め、pにBのタグがついていれば、ｇ pを計算して、Cの要素を求めることを表している。

Quantifier

∀ を含む命題の証明のルール

- Formationルール

$$\frac{\begin{array}{c}[x : A]\\ \vdots\\ A\text{は命題である} \quad P\text{は命題である}\end{array}}{(\forall x{:}A).P \quad \text{は命題である}} \ (\forall \mathbf{F})$$

- Introductionルール

$$\frac{\begin{array}{c}[x : A]\\ \vdots\\ p : P\end{array}}{(\lambda x{:}A).p : \ (\forall x{:}A)\ P} \ (\forall \mathbf{I})$$

∀ を含む命題の証明のルール

- Eliminationルール

$$\frac{a : A \qquad f : (\forall x{:}A)\ P}{f\ a : P[a/x]}\ (\forall \mathbf{E})$$

- Computationルール

$$((\lambda x : A)\ .\ p)\ a = p[a/x]$$

∃ を含む命題の証明のルール

- Formationルール

$$\frac{\begin{array}{c}[x : A]\\ \vdots\\ A\text{は命題である} \quad P\text{は命題である}\end{array}}{(\exists x{:}A).P \quad \text{は命題である}} \ (\exists \mathbf{F})$$

- Introductionルール

$$\frac{a : A \qquad p : P[a/x]}{(a,p) : (\exists x{:}A)\ .P} \ (\exists \mathbf{I})$$

∃ を含む命題の証明のルール

- Eliminationルール

$$\frac{p : (\exists x{:}A).P}{\mathrm{Fst}\ p : A}\ (\exists E'_1) \qquad \frac{p : (\exists x{:}A).P}{\mathrm{Snd}\ p : A}\ (\exists E'_2)$$

- Computationルール

$$\mathrm{Fst}\ (p, q) = p \qquad \mathrm{Snd}\ (p, q) = q$$

Calculus of Inductive Constructions

数学的帰納法

- ある命題Pが全ての自然数で成り立つことを示すのに次の「数学的帰納法」を用いることができる。
  1. Pはゼロで成り立つ。
     P(0)
  2. もし、Pがnで成り立つなら、Pはn+1でも成り立つ。
     P(n) --> P(n+1)
  3. この時、全ての自然数nについてPは成り立つ。
     ∀n P(n)
- 数学的帰納法は、その名前に反して、数学的演繹の強力な手段である。

COQとInductive Type

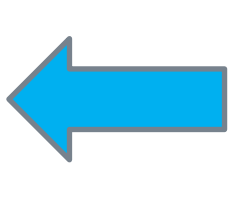
- 先にみたInductive Typeの例としてのnat (自然数)の型の次の定義は、COQのものである。

```
Inductive nat : Type :=
  | 0 : nat
  | S : nat -> nat.
```

- COQでは、全ての型は、関数の型かInductive Typeとして定義される。こうしたアプローチを CIC (Calculus of Inductive Constructions)と呼ぶ。

COQとInductive Type

- ただ、先のnatのInductiveとされる定義が、recursiveな定義であることはすぐわかるのだが、それと数学的帰納法のInductiveとの関係は、自明ではない。
- 実は、COQでは、Inductive Type **x** が定義されると、次の名前を持つ三つの型が、内部で自動的に定義される。
  - **x_ind**
  - **x_rect**
  - **x_rec**
- Inductive Type natの場合は、次のような三つの型が定義されることになる。
  - **nat_ind**
  - **nat_rect**
  - **nat_rec**



```
Inductive nat : Type :=
  | 0 : nat
  | S : nat -> nat.
```

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

```
nat_rect
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

```
nat_rect
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

nat上で定義されTypeに型を持つ全てのPについて

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

```
nat_rect
     : ∀ P : nat → Type,
       P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

nat上で定義されTypeに型を持つ全てのPについて
P O が成り立ち

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

```
nat_rect
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

nat上で定義されTypeに型を持つ全てのPについて
P O が成り立ち
natに属する全てのnについて、P n ならば P (S n)) が成り立つなら

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

```
nat_rect
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

nat上で定義されTypeに型を持つ全てのPについて
P O が成り立ち
natに属する全てのnについて、P n ならば P (S n)) が成り立つなら
natに属する全てのnについて、P nが成り立つ、

nat_rect はどういう型か？

- nat_rect はどういう型か、COQのコマンド Checkで見てみよう。

```
Check nat_rect.
```

nat_rect の型をチェックする

```
nat_rect
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

nat上で定義されTypeに型を持つ全てのPについて
P O が成り立ち
natに属する全てのnについて、P n ならば P (S n)) が成り立つなら
natに属する全てのnについて、P nが成り立つ、

これは、数学的帰納法に他ならない！

nat_ind と nat_rec の定義

- nat_ind と nat_rec の定義を見ておこう。

```
nat_ind =
fun P : nat → Prop ⇒ nat_rect P
     : ∀ P : nat → Prop,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

```
nat_rec =
fun P : nat → Set ⇒ nat_rect P
     : ∀ P : nat → Set,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

nat_ind と nat_rec の定義

□ nat_ind と nat_rec の定義で違うところ。

```
nat_ind =
fun P : nat → Prop ⇒ nat_rect P
     : ∀ P : nat → Prop,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

```
nat_rec =
fun P : nat → Set ⇒ nat_rect P
     : ∀ P : nat → Set,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

nat_ind と nat_rec の定義

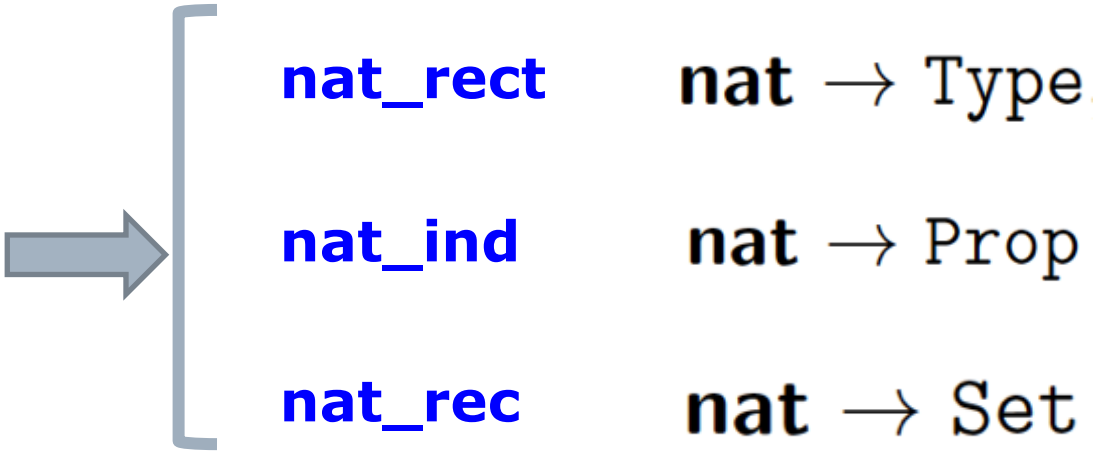
□ nat_ind と nat_rec の定義で同じところ。

```
nat_ind =
fun P : nat → Prop ⇒ nat_rect P
     : ∀ P : nat → Prop,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

```
nat_rec =
fun P : nat → Set ⇒ nat_rect P
     : ∀ P : nat → Set,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

nat_rect と nat_ind と nat_rec
Type と Prop と Set

```
Inductive nat : Type :=
  | 0 : nat
  | S : nat -> nat.
```



Type と Prop と Set は、異なる型である。

その違いは、COQでは大きな意味を持っているが、
その説明は、ここでは割愛する。

Homotopy Type Theory

新しい型の理論 HoTT

- Homotopy Type Theory (HoTT) は、数学者 Voevodsky が中心となって構築した、新しい型の理論である。
- HoTTは、数学の一分野であるホモトピー論やホモロジー代数と、論理学・計算機科学の一分野である型の理論とのあいだに、深い結びつきがあるという発見に端を発している。
- ここで取り上げられている型の理論は、Martin-LöfのDependent Type Theoryである。

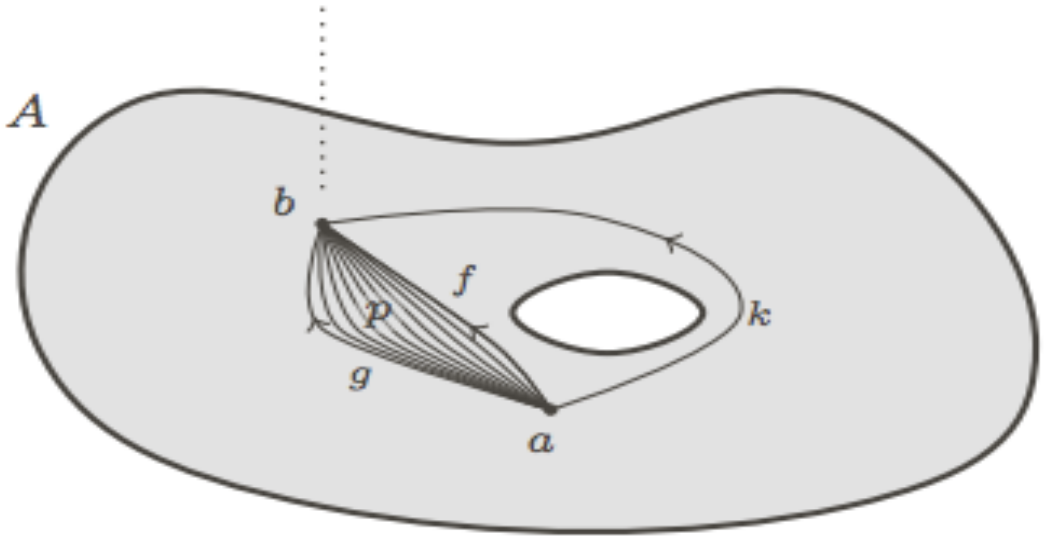
HoTTでの ***a : A*** の解釈

- 論理＝数学的には、**a : A** には、様々な解釈がある。ここでは、他の解釈と比較して、HoTTでの **a : A** の解釈を見てみよう。

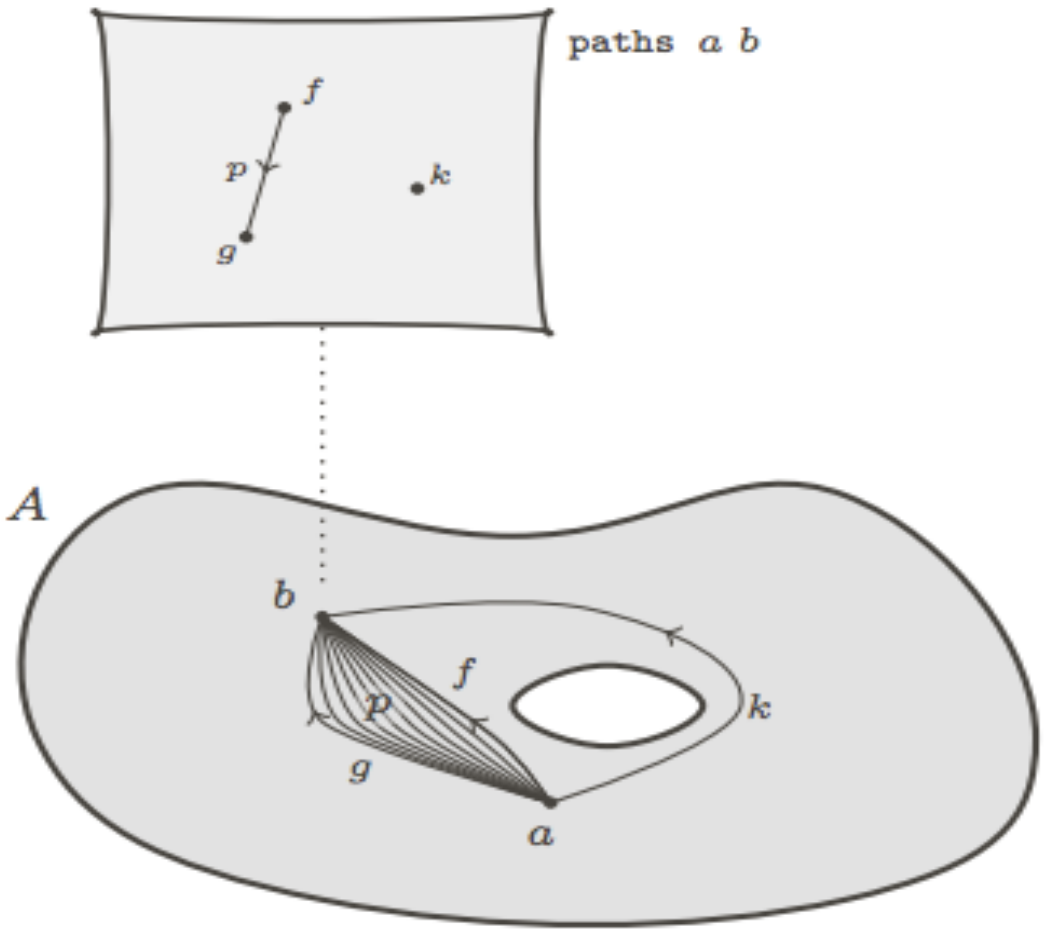
1. 集合論: Russellの立場
   Aは集合であり、aはその要素である。
2. 構成主義: Kolmogorovの立場
   Aは問題であり、aはその解決である
3. PAT: Curry & Howardの立場
   Aは命題であり、aはその証明である。
4. **HoTT: Voevodskyの立場**
   Aは空間であり、aはその点である。

HoTTでの「同一性」の解釈

空間Aの中で、
点aと点bをつなぐ「道」がある時、
aとbは、同じものと見なす。



HoTTでの同一性の解釈



空間Aの中で、
点aと点bをつなぐ「道」がある時、
aとbは、同じものと見なす。
「道」自体は、連続的に変化しうる。
その同一性は、
homotopyが
与える

Pathによる同一性の解釈

Equality	Homotopy	∞-Groupoid
reflexivity	constant path	identity morphism
symmetry	inversion of paths	inverse morphism
transitivity	concatenation of paths	composition of morphisms

Pathによる同一性の解釈

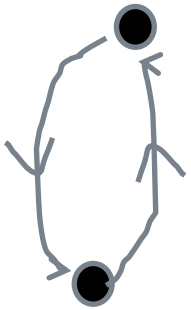
Equality	Homotopy	∞-Groupoid
reflexivity	constant path	identity morphism
symmetry	inversion of paths	inverse morphism
transitivity	concatenation of paths	composition of morphisms

reflexivity

a = a

自分自身に
帰る道

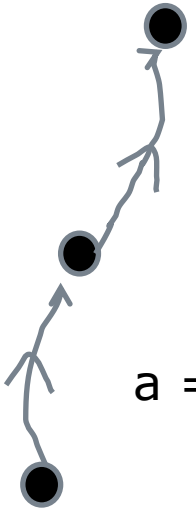
symmetry



a = b → b = a

逆の道

transitivity



a = b & b = c → a = c

道の結合

Pathによる同一性の解釈

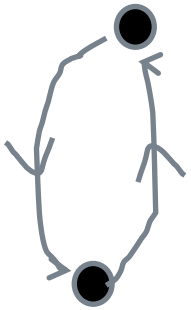
Equality	Homotopy	∞-Groupoid
reflexivity	constant path	identity morphism
symmetry	inversion of paths	inverse morphism
transitivity	concatenation of paths	composition of morphisms

reflexivity

a = a

自分自身に
帰る道

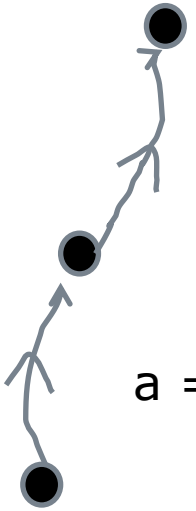
symmetry



a = b → b = a

逆の道

transitivity



a = b & b = c → a = c

道の結合

Pathによる同一性の解釈

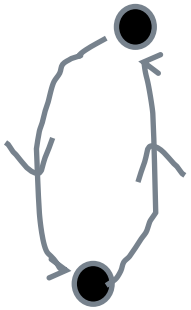
Equality	Homotopy	∞-Groupoid
reflexivity	constant path	identity morphism
symmetry	inversion of paths	inverse morphism
transitivity	concatenation of paths	composition of morphisms

reflexivity

a = a

自分自身に
帰る道

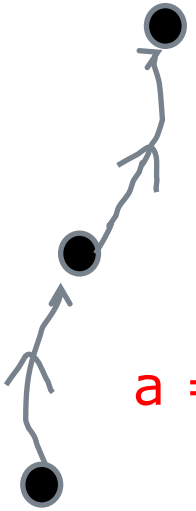
symmetry



a = b → b = a

逆の道

transitivity



a = b & b = c → a = c

道の結合

HoTTでのDependent Typeの解釈

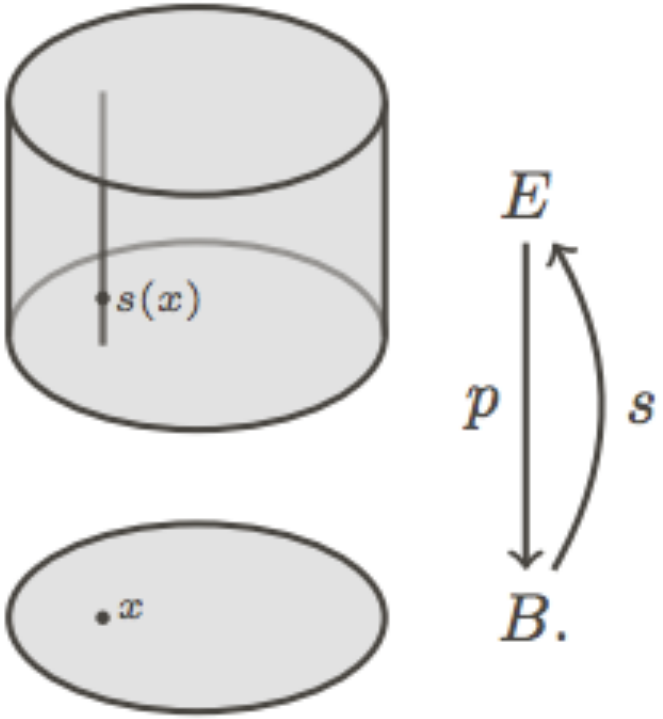
$(E_x)_{x \in B}$

Bの値で、
パラメータ
ー化されたE

HoTTでのDependent Typeの解釈

$(E_x)_{x \in B}$

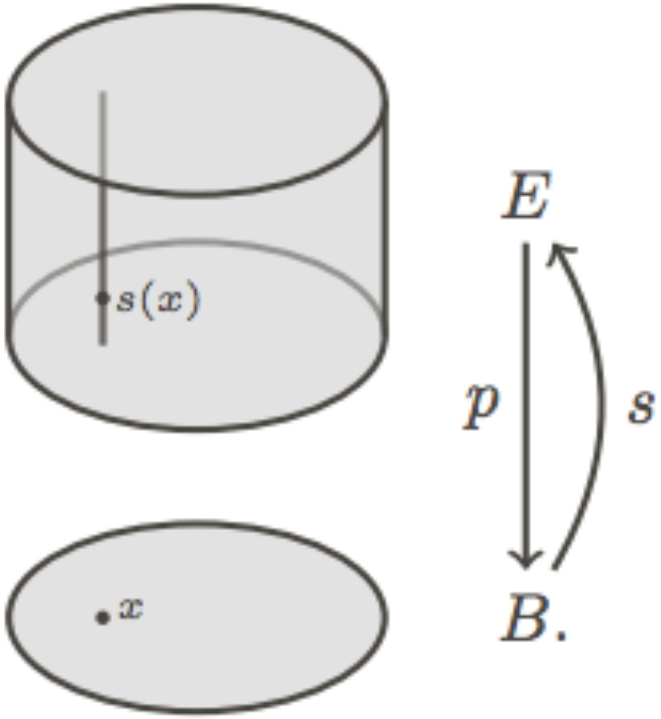
Bの値で、
パラメータ
ー化されたE



HoTTでのDependent Typeの解釈

$(E_x)_{x \in B}$

Bの値で、
パラメータ
ー化されたE



type theory		*homotopy theory*
$(x : B)$	$E(x)$	$p : E \to B$ is a fibration over B
$(x : B)$	$s(x) : E(x)$	s is a section of p

Univalent TheoryとCoq

- Voevodskyは、HoTTを武器に、Univalent Theoryという数学の新しい基礎付け・再構成を始めている。興味深いのは、彼が、こうした理論展開を、数学論文ではなく、Coqのプログラムの形でGitHubで公開していることである。
  https://github.com/vladimirias/Foundations/
- 次の論文は、型の理論やHoTTにあまりなじみのない一般の数学者向けの、Voevodskyの理論 Coqライブラリーの解説である。
  http://arxiv.org/pdf/1210.5658v1.pdf

VoevodskyとUniMath

- 一昨年（2017年）亡くなったVoevodskyは、Milner予想、Bloch-Kato予想を解くなど、代数幾何でグロタンディックが進もうとした道で、大きな業績を残した。Voevodskyの最後の仕事は、数学の基礎とコンピュータに関係していた。
- 彼は、数学の証明に、コンピュータを使うべきだと主張した最初の数学者の一人で、また、そのためのコンピュータによる証明支援システムのライブラリーUniMathを開発した。
  GitHub: https://github.com/UniMath/UniMath
- 2016年9月の講演 "UniMath - a library of mathematics formalized in the univalent style" https://goo.gl/3sJr1M

21世紀の数学の形式

- Voevodskyは、考える。「数学が、累積的（accumulative）な性格を持つのなら、もしも、そこに誤りが紛れ込むと、それも、累積する可能性がある。」
- ワイルズのフェルマーの定理の1993年の証明には、誤りがあった。それが修正されたのは、1995年のことだ。どんどん複雑化して高度化する数学の「証明」の正しさをチェックするのは難しい。数学者の「証明」が正しいという保証はないのだ！
- Voevodskyは、数学の証明は、コンピュータでチェックできるプログラムの形を取るべきだと主張し、実際に、それを実行してみせた。
- この流れは、21世紀の数学の形式を、大きく変えていくだろう。