『計算論理学』講義資料

亀山幸義

筑波大学 情報学群 情報科学類

http://logic.cs.tsukuba.ac.jp/~kam/complogic/

目次

1 はじめに

通常、「計算」と「論理」は、動的/静的として対極にあるもの (相補的なもの) と考えられている。

計算 (Computation)	動的 (dynamic)、いかに変化するか
論理 (Logic)	静的 (static), いかに変わらない性質を記述するか

表 1 「計算」と「論理」に関する普通のイメージ

このような理解でも第一近似としては間違いとは言えないが、実は、計算にも静的な側面があり、論理にも動的な側面がある。すなわち、静的/動的ということと計算/論理ということは独立である。

	動的な側面	静的な側面
計算	普通の計算	?
論理	?	普通の論理

表 2 「計算」と「論理」に関する改善したイメージ

「普通の計算」とは、通常のプログラミング言語で記述されるようなプログラムの世界である。「普通の論理」とは、通常我々が目にする論理体系で記述される。たとえば、ソフトウェアの正しさを保証するための Hoare 論理[*1]などの体系はこの表でいう「普通の論理」に該当する。普通の論理はあくまで静的であり、プログラムの実行前に正しさを保証するため等に用いられる。

この講義では、この表の?のところにも登場人物が存在することを示す。すなわち、?のところを埋めることによって、実は、計算体系と論理体系は本質的に同じものであることを理解することを目的とする。

プログラム言語論は、プログラム言語に関する科学である。その中心は、特定のプログラム言語に依存した個別の議論ではなく、プログラム言語によらない一般的な仕組みについての理論の展開である。また、単に個別のプログラム言語の機能等を比較するのではなく、多数のプログラム言語に共通する機能の本質について考察する学問である。

プログラム言語によらない一般的仕組み、共通の機能について議論するためには、個別の機能をそぎ落としたシンプルなプログラム言語を用意するのがよい。ここでは、対象を明確にするための非常に単純なプログラミング言語として、型付きラムダ計算 (typed lambda calculus) の体系を取りあげて、それについて講義を行う。型付きラムダ計算は、「計算」の中心をなす機構を抽象したラムダ計算に、「型」の概念を導入したものである。現代的なプログラム言語の多くは洗練された型の概念を持つため、型付きラムダ計算の拡張ととらえることができる。したがって、本講義で述べる理論や技法を、一般のプログラム言語に拡張することが可能である。

一方、本講義で扱う論理は、普通の論理 (古典論理という) だけではなく、直観主義論理という (普通の論理とは若干異なる) 論理も扱う。直観主義論理は、構成的論理あるいはプログラムの論理とも呼ばれ、計算との

[*1] Hoare 論理は、コンピュータ科学界のノーベル賞と言われるチューリング賞受賞者である C. A. R. Hoare が創始した、プログラム検証のための論理である。詳細は、「プログラム理論」の授業か、文献 (たとえば、林晋著「プログラム検証論」共立出版) を参照すること。

相性の良い論理である。普通の論理は、直観主義論理に対する拡張と考えることができる。

本講義では、型付きラムダ計算の静的および動的な側面、直観主義論理の静的および動的な側面について解説し、それらが本質的に同じものであることを理解する。また、種々の拡張についても触れ、現代的プログラム言語や「普通の論理」をどのように扱うことが可能かについても概観する。

1.1 参考書

本講義の内容については講義ノートを参考にしてほしい。講義内容よりさらに進んだ事を勉強するためには、下記書籍が参考になる (ここでは日本語の書籍のみをあげた)。

- 型のないラムダ計算・・・高橋正子「計算論」(近代科学社)
- 型付きラムダ計算・・・大堀淳「プログラム言語の基礎理論」 (共立出版), 五十嵐淳「プログラミング言語の基礎概念」 (サイエンス社)
- 記号論理学・・・小野寛晰「情報科学における論理」(日本評論社)、萩谷昌己「ソフトウェア科学のための論理学」(岩波講座)
- 形式化・・・佐藤雅彦、桜井貴文「プログラムの基礎理論」(岩波講座)

2　ラムダ記法と (型のない) ラムダ計算

2.1　ラムダ記法

ラムダ記法 (lambda notation) は，関数の表記において，仮引数となる変数を明示した記法である．これにより，関数と，その関数に引数を与えた計算結果 (値) の区別がつくようになる．

例 1　数学の本での記法: $f(x) = ax^2 + bx + c$　これでは $f(x)$ が関数なのか f に x を食わせた結果の値なのかわからない．もし，関数とデータ (自然数や実数など) が，いつでも文脈から区別できるのであれば，このような曖昧さがあっても構わないが，コンピュータ科学では，関数やプログラム自身を扱う関数 (高階関数，メタプログラム) が平気で現れる．そのような場合には，文脈からだけでは，関数なのかデータなのかわからない．

ラムダ記法での関数 f: $f = \lambda x.ax^2 + bx + c$

ラムダ記法での値 $f(x)$: $f(x) = (\lambda x.ax^2 + bx + c)x = ax^2 + bx + c$

f は関数であり，$f(x)$ は値 (関数 f に x を引数として食わせた結果として得られる値) であり，両者は明確に区別される．

ラムダ記法を使うと高階の関数 (higher order function) も記述することができる．高階の関数とは，引数や返り値が関数であるような (高いレベルの) 関数である．

例 2　高階関数:

$double = \lambda f.(\lambda x.f(f(x)))$ … 関数 f とデータ x を引数としてもらい，f を x に 2 回適用した値を返す高階関数

$compose = \lambda f.(\lambda g.\lambda x.g(f(x)))$ 関数 f と関数 g を引数としてもらい，f と g を合成した関数を返す．

たとえば，$f = \lambda x.x * 2$, $g = \lambda x.x + 1$ とするとき，$double(f)$ は，$\lambda x.x * 4$ を表し，$double(g)$ は $\lambda x.x + 2$ を表し，$compose(f, g)$ は $\lambda x.x * 2 + 1$ を表す．

なお，ラムダ記法の伝統に従い，$f(x)$ という表記の括弧を省略して fx と書くことにする．もちろん，括弧が必要になればいくらでも補うこととする．

2.2　構文

(型のない) ラムダ計算の体系を定義する．本章は，型付きラムダ計算への導入であるため，厳密な形式的定義は省略して，自然言語で非形式的に述べるに留める．

まず，変数 $x, y, z, \cdots$ が無限個あるとする．また，定数 $c, d, \cdots$ が有限個もしくは無限個あるとする．ここで，何が定数になるかはプログラム言語に依存して決まるので，ここでは特に定めない．そのとき，ラムダ計算の項 (term, ラムダ項，ラムダ式ともいう) は以下で定義される．

定義 1　[型なしラムダ計算の項]

- x が変数であれば，x は項である．
- c が定数であれば，c は項である．
- M と N が項であれば，(MN) は項である．(このような項を application (適用) という．)

- x が変数であり，M が項であれば，$(\lambda x.M)$ は項である．(このような項を abstraction (抽象) という．)

たとえば，$(\lambda x.(cx))$ や $((xy)(\lambda z.c))$ は項である．

このように記述すると、括弧が大量に必要となり読みにくいので、一定のルールのもとで、括弧を省略する。

まず、一番外側の括弧はいつでも省略できる。つまり、xy は (xy) のことである。$\lambda x.M$ においては，M としてなるべく大きな項 (広い範囲) を取るようにする．つまり，$\lambda x.cx$ は $\lambda x.(cx)$ のことであって $(\lambda x.c)x$ ではない．後者の項を表現するためには，括弧は省略できない．

また，LMN のように，3 個以降の項が適用の形で並んでいるときは左から括弧をつけることにする．つまり，この項は $(LM)N$ をあらわすのであって，$L(MN)$ ではない．

例 3　括弧が省略された式:

$\lambda x.yz\lambda u.vxy$ は括弧を補うと，$\lambda x.((yz)(\lambda u.((vx)y)))$ のことである．

2.3　変数の束縛と α 同値

ラムダ計算や論理の体系を最初に習うときに，最も理解しにくいのが変数の束縛の概念である．

$\lambda x.(\lambda x.x)$ という関数 f において，一番右の x は，どちらの λ に対応するか考える．ラムダ計算の約束事では変数に対応する λ は，その変数を囲む最も内側の λ とする．したがって，f は $\lambda y.(\lambda x.x)$ と同じ関数であって，$\lambda x.(\lambda y.x)$ とは異なる関数である．この「同じ関数」、「異なる関数」という概念をきちんと考えることにする．

一般に項 M は、変数 x を 2 回以上含むことがある。それらを区別するため「項 M における x の 2 回目の出現」等という。ただし、λ の直後の変数については、「出現」と考えない。たとえば、$\lambda x.\ x\ (\lambda y.\ x)$ には、x という変数は 3 回出てくるが、最初のものは λ の直後にあるのでカウントせず、この項に x は 2 回出現するという。

「出現」という言葉を出したのは、項における位置関係により、変数の扱いが異なるからである。たとえば、たとえば、$\lambda x.\ x\ (\lambda x.\ x)\ x$ は x の出現を 3 つ含むが、1 つ目と 3 つ目の x の出現は、最初の λ と対応付き、2 番目の x の出現は、2 つ目の λ と対応付くことが (直感的には) 理解できるであろう。このような λ と変数の出現の対応関係を「束縛」と言う。「束縛」をもう少し精密に定義しよう。

項 M における変数 x の出現は、その x を囲む $\lambda x.$ のうち、最も近くにある $\lambda x.$ によって束縛される、と言う。どの λ でも束縛されない出現を，自由な出現という．例えば，$\lambda x.\ (\lambda y.\ x\ y)\ y$ は、x の出現を 1 つ、y の出現を 2 つ持つが、x の 1 番目の出現 は λx で束縛され，y の 1 番目の出現 は λy で束縛され，y の 2 番目の出現 は自由である．

項 $\lambda x.M$ に対して、M の中の x の自由な出現 (M で自由な出現は、$\lambda x.M$ では一番外の $\lambda x.$ で束縛される) をすべて一斉に別の変数 y で置き換えたものを M' とする。ただし、y は M において自由な出現を 1 つも持たない変数 (M に現れないか、現れたとしても束縛された出現しか持たない) とする。このとき、$\lambda x.M$ と $\lambda y.M'$ は実質的に同じ関数を表しており、特に α 同値であるという。

項 M に対して (M の一部に対して) 上記の置き換えを 0 回以上繰返して項 N が得られるとき、M と N も、α 同値であるといい、$M \equiv_\alpha N$ と書く．例えば，$\lambda x.\ (\lambda y.\ x\ y)\ y$ と $\lambda z.\ (\lambda y.\ z\ y)\ y$ と $\lambda z.\ (\lambda x.\ z\ x)\ y$ とは、いずれも α 同値である。ここで、最初の項で使われている x を最後の項では、別の意味で使っている

ことに注意されたい。このようなものも α 同値である。一方で、上記の 3 つの項と、$\lambda y.\ (\lambda z.\ y\ z)\ y$ とは α 同値ではない。これ以降、α 同値である 2 つの項は同一視する[*2]．なお，このテキストではときどき $M \equiv N$ と書くことがあるが，これは M と N が (括弧を省略せずに書いたときに) 文字列として完全に一致するときのことである．当然ながら，$M \equiv N$ であれば $M \equiv_\alpha N$ であるが，逆は必ずしも言えない．

なお，α 同値性は，変数の名前換えだけで形式的に一致する項の間の関係であり，$\lambda x.x * 2$ と $\lambda x.x + x$ のように意味的に一致するだけの場合は，α 同値とは呼ばない．また，$(\lambda x.x)1$ と 1 は次節で見るように，計算によって一致するが、その場合も α 同値ではない．後者については $=$ という記号を使うことにする．すなわち，$(\lambda x.x)1 \equiv_\alpha 1$ でないが，$(\lambda x.x)1 = 1$ である．

2.4 計算規則

次に計算規則を与える．ラムダ計算は関数の概念だけを持つので，計算といっても，「関数に引数を与えると，その結果の値になる」という形のものだけである．

定義 2 [β-簡約]

> 項 L の中に $(\lambda x.M)N$ の形があるとき，それを $M\{x := N\}$ で置き換える操作を β-簡約と呼ぶ．β-簡約のことを $\rightarrow_\beta$ もしくは $\rightarrow$ と表記する．

ここで，$M\{x := N\}$ は、すぐ後で定義する。

項の中の $(\lambda x.M)N$ の形の部分項を，計算可能な部分項 (reducible expression)，もしくは，レデックス (基, redex) という．

β-簡約の定義に出てくる代入 (substitution) は以下のように定義される．

定義 3 [代入] 項 M の中の自由な x の出現に 項 N を代入して得られる項 $M\{x := N\}$ は以下のように定義される項である．

- $x\{x := N\} \equiv N$
- $y\{x := N\} \equiv y$, ただし x と y が異なる変数のとき
- $(LM)\{x := N\} \equiv (L\{x := N\})(M\{x := N\})$
- $(\lambda y.M)\{x := N\} \equiv \lambda z.((M\{y := z\})\{x := N\})$, ただし,$z$ は M, N に自由な出現を持たない変数

代入の定義の最後のケースが非常にややこしいが，これの詳細と改善策については型付きラムダ計算の章で述べることにして，ここでは，例を与えるのみとする．

$$
\begin{aligned}
(\lambda x.\lambda x.x)1 &\rightarrow (\lambda x.x)\{x := 1\} \\
&\equiv \lambda z.(x\{x := z\})\{x := 1\} \\
&\equiv \lambda z.z\{x := 1\} \\
&\equiv \lambda z.z \\
&\equiv_\alpha \lambda x.x
\end{aligned}
$$

[*2] 同一視するとはいっても，見た目に違う項であるのでそれらを同じと見るためには多少の訓練が必要である

$$
\begin{aligned}
(\lambda x.\lambda y.xy)y &\to (\lambda y.xy)\{x := y\} \\
&\equiv \lambda z.(xy\{y := z\})\{x := y\} \\
&\equiv \lambda z.(xz)\{x := y\} \\
&\equiv \lambda z.yz
\end{aligned}
$$

2.5 合流性と計算戦略

ラムダ計算の計算規則は，非決定的 (non-deterministic) である．すなわち，項 M を計算するやり方が 2 通り以上あることがある．たとえば，$(\lambda x.x+1)((\lambda y.y+2)3)$ という項は，以下の 2 つのレデックスを持つ．

$$
\begin{aligned}
\underline{(\lambda x.x+1)((\lambda y.y+2)3)} &\to ((\lambda y.y+2)1)+1 \\
(\lambda x.x+1)(\underline{(\lambda y.y+2)3}) &\to (\lambda x.x+1)(3+2)
\end{aligned}
$$

現実のプログラミング言語は，通常，決定的な計算規則をもつ[*3]ので，ラムダ計算を現実のプログラミング言語のモデルと考える場合には，適用可能な計算規則を制限する必要がある．適用できる計算規則を定めることを「計算戦略 (strategy) を定める」という．

ここでは，最も重要な 2 つの計算戦略のみをあげる．

定義 4 [値呼び戦略, call-by-value] 関数呼びだし $(\lambda x.M)N$ の計算において，まず，実引数 N を計算して，次に，その計算結果である v を M に代入して，項 $M\{x := v\}$ を作り，最後にこの項の計算を行い，結果を得る．

定義 5 [名前呼び戦略, call-by-name] 関数呼びだし $(\lambda x.M)N$ の計算において，実引数 N を計算せずに，項 $M\{x := N\}$ を作り，この項の計算を行い，結果を得る．

計算結果のことを「値 (value)」と呼ぶので，第一の戦略は値呼びといわれる．第二の戦略は，$M\{x := N\}$ の計算において，N という実引数を (x という) 名前で参照するので，名前呼びといわれる．

計算の効率の面からいうと，値呼び戦略も名前呼び戦略も一長一短であり，どちらが常に優れている，ということはない．そこで，両者の長所をとった必要呼び戦略 (call-by-need) というものがある．これは，名前呼び戦略に似ているが，$M\{x := N\}$ という代入を行わずに (代入をしてしまうと，N が たくさんコピーされる可能性がある)，実際には N へのポインタをばらまくものである．N の計算が何回も呼ばれる可能性があるが，最初に呼ばれたときに N へのポインタの先を N の計算結果に置きかえてしまえば，2 回目からは結果を拾うだけでよく，計算が高速になる．

[*3] これは必ずしも正確ではない．プログラミング言語の仕様書の中には，「どちらであるか決めない」ということにより，非決定的な計算規則を許すものがある．たとえば，C 言語の仕様書では，関数の実引数を計算する順番は決まっていない．したがって，`foo(a,b)` という関数呼び出しにおいて，a と b ともに副作用をもつ計算の場合，どちらを先に計算するかによって結果は異なることがある．また，並列プログラミング言語は必然的に非決定的である．

2.6 再帰定理

ここまでのところ，型のないラムダ計算の体系は大変シンプルであるため，つまらない体系のように思える．実際，現代的プログラム言語はどれも，ラムダ計算よりはるかに豊富で複雑な構文，計算規則をもっている．

次の定理は，その予想に反して，「計算」という観点では，型のないラムダ計算はとんでもない力を秘めていることを示している．

まず、部分関数 $f;\ A \to B$ とは、A のすべての要素に対して、B の要素 1 つを対応付けるかあるいは、1 つも対応付けない (未定義) のいずれかであるような対応付けである。関数 $f : A \to B$ は、A のすべての要素に B の要素 1 つを対応付けるので、関数と部分関数の違いは、後者は、未定義を許すことである。たとえば、$f(x) = 1/x$ は、実数から実数への部分関数であるが、関数ではない。($f(0)$ は未定義である。) この f は、正の実数から実数への関数であり、部分関数でもある。部分関数を、関数と区別して、$f;\ A \to B$ と書くことがある (コロンのかわりにゼミコロンを使う。)

定理 1（計算可能関数） $\mathcal{N}$ を自然数の集合とし，$\mathcal{N}^n$ を $\mathcal{N}$ の n 個の直積とする．

部分関数 $f;\mathcal{N}^n \to \mathcal{N}$ に対して，以下の条件は全て同値である．

- f はチューリング機械 (Turing Machine) で表現可能である．
- f は partial recursive function (帰納的部分関数) である．
- f は型のないラムダ計算で定義可能である．

チューリング機械は，非常に原始的なコンピュータ (というより，プログラム言語) であるが，現代的な高性能コンピュータと同等の計算能力を持っている．したがって，上記の定理は，ラムダ計算も現存するあらゆる高性能なプログラム言語と同等の計算能力を持っていることを示している．

今日のコンピュータ科学の最重要の基礎の 1 つである，Church の提唱 (Church's Thesis, Church-Turing の提唱ともいわれる) は，計算可能 (部分) 関数とは，上記のいずれかで定義される (部分) 関数のことと定義しよう，というものである．

上の定理は驚くべきものであるが，その証明の詳細をここで示すことはできない．そのかわりに，証明の鍵となる定理を述べておく．

定理 2（再帰定理 (Recursion Theorem)） 型なしラムダ計算において，再帰呼び出しは常に解を持つ．すなわち，

どんな項 F と変数 x に対しても，$fx = Ffx$ が成立する項 f を作ることができる．

ここで $=$ は β-簡約によって一致させることができる，という意味である．

再帰定理の証明のためには

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

というラムダ項 (Curry の不動点演算子, Y-combinator) を考えればよい．すると，

$$\begin{aligned} YFx &\equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))Fx \\ &\to (\lambda x.F(xx))(\lambda x.F(xx))x \\ &\to F((\lambda x.F(xx))(\lambda x.F(xx)))x \\ F(YF)x &\equiv F((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))F)x \\ &\to F((\lambda x.F(xx))(\lambda x.F(xx)))x \end{aligned}$$

となり，YFx と $F(YF)x$ が β-簡約によって一致することがわかった．よって、どんな F に対しても YF という項を考えれば、$fx = Ffx$ の解 f となる。

例 4　自然数の階乗を求める関数 f は，Scheme 言語では，以下のように「定義」される．

```
(define (f n)
  (if (= n 0) 1
    (* n (f (- n 1)))))
```

これは、数学的には「定義」ではない。なぜなら、f の定義の中に f が現れるからである。しかし、プログラミング言語ではこのようなものも「定義」と見なしている。そこで、なぜ、これが「f の定義」と見なしてよいかを再帰定理を使って説明しよう。

上記の「定義」は、ラムダ計算では，以下のような方程式としてあらわせる。

$$\mathtt{f}n = (\lambda f.\lambda n.\mathtt{if}(n = 0,\ 1, n * (f(n-1))))\mathtt{f}n$$

ここで、if-then-else を if(a,b,c) の形で書いた。

$F = \lambda f.\lambda n.\mathtt{if}(n = 0,\ 1, n * (f(n-1)))$ とおくと、上記の方程式は、$\mathtt{f}x = F\mathtt{f}x$ の形をしている。再帰定理により，この方程式は YF という解を持つ．すなわち，上記の Scheme の「定義」が、YF という項を定義していると思えばよいことになる。(厳密にいうと、再帰定理は、YF が解の 1 つになっていることを主張しているだけなので、上記の Scheme の定義が YF を定義しているつもりなのか、他の解を定義しているつもりなのかは定められないが、少なくとも、定義されるものが (1 つは) 存在する、ということは保証される。)

このようにして、再帰呼び出しによる関数定義 (伝統的な数学の範囲では、定義になっているかわからないもの) は、ラムダ計算の中で、再帰定理の形で、きちんと捉えることができる。

[2015/10/5 追加] 上記のテキストにおいて、Scheme 言語との関係は、やや不備があった。というのは、上記の Y という不動点演算子を、実際のプログラム言語 (Lisp や Scheme) でそのままプログラムとして記述しても、停止しないプログラムになってしまうからである。実際の Scheme では、上記の f という関数は、非負の整数に対して、その階乗を返してくれるので、停止しないプログラムを定義しているとは思えない。

この原因は、実際のプログラム言語は、「値呼び」という特定のやりかたで計算するためである。上記の再帰定理の証明では、値呼びとは別の (ある特定の) 順番に計算を進めたが、これは Scheme とは異なる順番であった。

そこで、Scheme などの値呼び計算の場合に利用できる不動点演算子というものが考えられる。これを Y_{cbv} と書くことにすると、次のように与えられる。

$$Y_{cbv} = \lambda f.(\lambda x.\lambda y.f(xx)y)(\lambda x.\lambda y.f(xx)y)$$

先ほどの Y とは少しだけ異なっており、これを Scheme や Lisp のプログラムとして記述すれば、きちんと (計算が止まるべきときには、ちゃんと止まる) 再帰関数を与える。

Scheme の一種である MzScheme (現在の Racket) 処理系で、Y_{cbv} と 上記の階乗関数を走らせた例を以下に載せる。

```
% mzscheme
Welcome to Racket v5.2.1.

(let ((y (lambda (f)
           ((lambda (x) (f (x x)))
            (lambda (x) (f (x x)))))
      )
      (factorial_gen (lambda (f) (lambda (x)
         (if (= x 0) 1 (* x (f (- x 1)))))))
      )
  ((y factorial_gen) 5))

^C (not terminating)
>
(let ((y_cbv (lambda (f)
               ((lambda (x) (lambda (y) ((f (x x)) y)))
                (lambda (x) (lambda (y) ((f (x x)) y)))))
      )
      (factorial_gen (lambda (f) (lambda (x)
          (if (= x 0) 1 (* x (f (- x 1)))))))
      )
  ((y_cbv factorial_gen) 5))

120
```

1 つ目の実行で^C とあるのは、実行が停止しないため、ctrl-C で停止した、という意味である。一方で、2 つ目の実行では Y_{cbv} を使っているので、きちんと計算が停止し、5 の階乗である 120 が答えとして返ってきている。

3　構文の定義方法 (BNF と帰納的定義)

プログラムや論理式などの文法 (構文) を定義するため，BNF (Backus Normal Form, Backus-Naur Form) による方法がよく使われる．

例 5（簡単なプログラム言語のプログラム）

```
<プログラム> ::= <宣言部> <関数定義部>
<宣言部> ::= "" | <宣言> ";" <宣言部>
<関数定義部> ::= "" | <関数定義> ";" <関数定義部>
<関数定義> ::= <型宣言> <関数名> "(" <引数部> ")" <ブロック>
<ブロック> ::= "{" <宣言部> <文の列> "}"
...
```

BNF はコンパクトな表現で、無限集合を定義できるという利点があるが，比較的単純な文法しか表現できない．

BNF による定義法を含む，より一般的な定義方法として帰納的定義 (inductive definition) がある．

例 6　葉が自然数のラベルを持つ 2 分木に対する帰納的定義

$$\frac{n : \text{自然数}}{\texttt{leaf}(n) : 2\text{ 分木}}\ \texttt{leaf} \qquad \frac{T1 : 2\text{ 分木} \quad T2 : 2\text{ 分木}}{\texttt{node}(T1, T2) : 2\text{ 分木}}\ \texttt{node}$$

この定義は，leafと nodeの 2 つの定義節から構成される．leaf規則は，「n が自然数であれば $\texttt{leaf}(n)$ が 2 分木である」ということを意味し，「初めの要素」を導入するものである．node規則は，「$T1, T2$ が 2 分木であれば，$\texttt{node}(T1, T2)$ が 2 分木である」ということを意味し，「ある要素から次の要素を作る操作」を導入するものである．

例: 2 分木 (葉が自然数のラベルを持つもの) の帰納的定義

$$\frac{n : \text{自然数}}{\texttt{leaf}(n) : 2\text{ 分木}}\ \texttt{leaf} \qquad \frac{T1 : 2\text{ 分木} \quad T2 : 2\text{ 分木}}{\texttt{node}(T1, T2) : 2\text{ 分木}}\ \texttt{node}$$

上記の 2 つのルールが 2 分木に対する帰納的定義は，(明示的には書かないが，いつでも) 以下のルールを含む．

上記の 2 つのルールを有限回適用してできたものだけが，2 分木である．

この最後のルールは，帰納法 (2 分木に関する構造帰納法) として表現される．

例 7　$P(T)$ を 2 分木 T に関する命題とする．すると，以下の帰納法が成立する．

$$\frac{\begin{array}{cc} n : \text{自然数} & P(T1) \quad P(T2) \\ \vdots & \vdots \\ P(\texttt{leaf}(n)) & P(\texttt{node}(T1, T2)) \end{array}}{\forall T : 2\text{ 分木}.P(T)}\ \text{帰納法}$$

このルールは，「すべての 2 分木 T に対して P という性質が成立することを証明するためには，(1) n が自然数であるという仮定のもとで $\texttt{leaf}(n)$ に対して P が成立すること，(2) $T1$ と $T2$ に対して P が成立するという仮定のもとで $\texttt{node}(T1, T2)$ に対して P が成立すること，の 2 つを証明すればよい，というものである．これが，自然数に対する数学的帰納法の拡張になっていることは容易に想像できるだろう．

4　命題論理

本章では，最も基本的な論理として命題論理を取りあげ，その形式的体系について考察する．この論理体系と前章の計算体系との関係については次章で述べる．

4.1　命題

命題 (proposition) とは何か？

この問に対する通常の答えは，「命題とは，真か偽かが確定している文」である．あるいは「命題とは，真であるかどうかを考えることができる文」とも言えるだろう．実は，この答えはいつでも正しいとは限らず，どのような論理体系を考えるかによって変わってくるのであるが，当面は (普通の論理だけを考えている限りは) 良いだろう．たとえば，「日本で一番高い山」は命題ではなく，「日本で一番高い山は筑波山である」は (偽であるような) 命題であり，「日本で一番高い山は富士山である」は (真であるような) 命題である．

我々が日常の推論で使う命題はこのような単純なものだけではない．

「A君は怒られなければ勉強しない」のように複合的な命題がある．このような複合的な表現も，構成要素である「A君が怒られる」「勉強する」という命題の真偽が決まれば全体の真偽が決まるので，命題であると言える．さらに，「A君が怒られなければ勉強しない」と「A君が勉強している」という 2 つの命題から「A君は怒られたはずだ」という推論をしたとする．これは「A君が怒られる (た)」「勉強する (した)」という命題の真偽にかかわらず正しい推論であると言えよう．論理学は，このように「個々の命題の真偽にかかわらず推論方法が正しいかどうかを議論する学問」ということができる．

本章では，命題に関する論理を扱う．

命題論理は，基本となる命題 (原始命題，基本命題，素命題，アトムなどという) が成立するかどうかに関係なく，いつでも成立する命題とは何か，それをどうやって推論するかについて考える．言い変えれば，基本命題の内部構造には立ち入らず，それらを組み合わせる記号 (論理記号) についての法則を考えるものである．

この観点に立って，命題を形式的に定義しよう．前述したように，基本命題の中身が何であるかは問わないので，それらを単に $K_1, K_2, \cdots, K_n$ という記号であらわす．このほかに，$\bot$ という特別な基本命題があるとする．これは「矛盾」 (どんな意味論でも「偽」となる命題) を表す[*4]．

これらをもとに，命題は以下のように帰納的定義により定義される．

定義 6　[命題]

- 基本命題 $K_1, \cdots, K_n$ は命題である．
- A, B が命題であるとき，$\neg A, A \wedge B, A \vee B, A \supset B$ は命題である．

$\neg$ は否定 (「でない」, not, negation)，$\wedge$ は論理積 (「かつ」，連言, and, conjunction)，$\vee$ は論理和 (「または」，選言, or, disjunction)，$\supset$ は含意 (「ならば」, imply, implication) を表す論理記号 (論理結合子) である．

複合的な命題の場合，括弧の付け方が問題になることがある．たとえば，$A \wedge B \supset C$ という命題は，$(A \wedge B) \supset C$ であるか $A \wedge (B \supset C)$ であるかわからない (曖昧さがある)．これでは厳密な考察の対象として

[*4] いつでも「真」となる命題は $\bot \supset \bot$ として作ることができるので，そのような記号は導入しない．

は問題があるので，括弧の補い方が一意的になるように，以下の規則を定める．

- 異なる論理記号の間の結合の強さは，$\neg$, $\wedge$, $\vee$, $\supset$ の順番とする．($\neg$ が一番強い．)
- 同じ論理記号の間の結合については，$\wedge$, $\vee$ は左結合的であり，$\supset$ は右結合的である．

たとえば，$\neg A \wedge B \wedge C \supset \neg D \supset E$ は，$(((\neg A) \wedge B) \wedge C) \supset ((\neg D) \supset E)$ をあらわす．

特に注意すべきは，$\supset$ が右結合的なことである．なぜそうなっているかはこの時点ではなかなかわからないが，あとで，論理と計算の対応を付けたときには「なるほど」と思うであろう．

4.2 形式と意味

コインの裏表のように，物事には形式と意味 (内容) がある．日常言語では「形式的」というのは否定的なニュアンスを伴い，「内容」というのは肯定的なニュアンスを伴う言葉であるが，論理学では必ずしもそうではない．

論理学における形式とは，形式的体系 (formal system) のことであり，これは帰納的定義等によって厳密に定められた構造である．直感や意味を排して決められるので，どんな人が読んでも同じ定義である．従って，コンピュータによっても処理することが可能である．人間はつい意味の世界に頼ってしまうが，厳密な推論をしたり (たとえば，数学者による証明)，コンピュータによる処理をする (たとえば，定理の自動証明) ときには，形式的体系が不可欠である．

一方，意味の世界は，意味論 (semantics) あるいはモデル論と呼ばれる．何らかの形式的体系を考える場合，通常は，それに先立って「何かあらわしたい世界」があるはずである．それを「意図した意味論」という．形式的体系を作る際の目標は，「形式的体系が，意図した意味論とちょうどぴったり対応する」ように作ることである．もちろん，ひとたび形式的体系を作ってしまえば，「意図した意味論」以外の意味論を持ち得ることは多い．たとえば，自然数の形式的体系は，我々が通常イメージしている自然数の世界を意図して作ったのであるが，「無限に大きな自然数」のようなものを含む「非標準的な意味論」を持つことが知られている．

形式的体系と意味論の関係を記述するのは以下の 2 つの性質である．

- 健全性: 形式的体系で証明 (導出) できることは，どんな意味 (モデル) のもとでも正しい．
- 完全性: どんな意味 (モデル) のもとでも正しいことは，形式的体系で証明 (導出) できる．

これら 2 つの性質が成立するとき，形式的体系と意味論は「ぴったり一致している」ということができる．

4.3 命題論理の意味論

命題論理の意味論は，通常，真理値表で与えられる．(真理値表による意味論しかないわけではない．また，真理値表の意味論とぴったり合っているのは古典論理だけである．この点は後に説明する．)

命題に対する真理値 (truth value) とは，「真」 (true, T) か「偽」(false, F) のいずれかの値のことである．形式化する前の命題は，真か偽かが定まっている文のことであったので，形式化した後でも (記号で表現した命題に対しても)，真であるか偽であるかを定めれば，意味が決まったことになる．

命題論理の真理値表の意味論は，以下のように与えられる．

A	B	$\neg\ A$	$A \wedge B$	$A \vee B$	$A \supset B$
T	T	F	T	T	T
T	F	F	F	T	F
F	T	T	F	T	T
F	F	T	F	F	T

真理値表については，学類 1 年生向け授業『離散構造』の資料[*5]に載っているので詳しくはそちらを参照してほしい．

命題 A に対応する真理値表の列が全部 T のとき，すなわち，A に含まれる基本命題の真理値がどのような値であっても A の真理値が T であるとき，A を恒真式 (tautology) と言う．恒真式とは「常に真である式」という意味であり，数学における「定理」に相当する．数学者がたくさんの美しい定理を追い求めているのと同様，論理学者は恒真式を追い求める．命題 A が恒真であることを，

$$\models A$$

と書くことがある．

命題 A が恒真であることを決定することは，工学的にも重要である．たとえば，プログラムやシステム設計にミスがないかどうかは，ある種の命題 (仕様を表す命題) が常に正しいかどうか，という問題に帰着される．真理値表意味論により，この問題が常に有限時間で判定可能であることがわかる．ただし，真理値表を本当に書くと，基本命題の数を k とするとき 2^k (2 の k 乗) の行数が必要になるため，大変に計算時間がかかる．次節で形式的体系を導入するが，これを使えば，(最悪の場合の計算時間は改善されないが，通常の場合には) 計算時間が大幅に短縮される．

4.4 形式的体系としての命題論理

一般的には，形式的体系を導入する意義として以下のものが考えられる．

- 人間の推論に近い形の推論を表現できる．
- コンピュータ上で推論を実行できる．(命題論理の場合は，前節で見たように，意味論をコンピュータでシミュレートすることが可能であったが，より複雑な論理ではそれは必ずしも可能ではない．)

これら以外に，この講義の題目は「計算」と「論理」の対応関係を調べることであり，このためには形式的体系は必須のものである．

さて，命題論理の形式的体系は，以下のように定義される．

まず，形式的な命題は前節で導入したように，基本命題と論理記号から構成されるものである．

判断 (judgement) は，$\Gamma \vdash A$ という形を取る．ここで，Γ は命題の有限列であり，たとえば，$K_1 \wedge (K_2 \supset K_1), K_3$ といったものである．Γ として空列も許す．Γ のことを宣言と呼んだり，文脈と呼んだりする．

以下の規則は，判断を木の形に組み上げるためのものであり，推論規則と呼ばれる．(木の 1 つ 1 つのノードが判断になっている．)

$$\frac{(\text{ただし } A \in \Gamma \text{のとき})}{\Gamma \vdash A}\ assume \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot E \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg I \qquad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash B}\ \neg E$$

[*5] http://logic.cs.tsukuba.ac.jp/ kam/discrete/, (第 1 章「命題と証明」)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge I \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge EL \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge ER$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee IL \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee IR \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee E$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ \supset I \qquad \frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \supset E \qquad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\ \neg\neg E$$

それぞれの規則の横棒の右隣に規則名を書いた．(assume, $\wedge I$ など)

ここで注目してほしいのは，上記の規則群がほぼ対称形であることである．すなわち，$\neg, \wedge, \vee, \supset$ の各記号に対応して，導入規則 (Introduction Rule), 除去規則 (Elimination Rule) がそろっていることである．ただし，例外は $assume, \bot E, \neg\neg E$ の 3 つの規則である．

上記の規則を有限回適用して $\Gamma \vdash A$ が推論できたときに「宣言 Γ のもとで命題 A は導出可能である (証明を持つ)」という．(Γ が空の列のとき，「命題 A は導出可能である」ということもある．)

例 8　命題論理の導出の例をあげる．

$$\dfrac{\dfrac{\overline{A \vdash A}\ assume}{\vdash A \supset A}\ \supset I}$$

$$\dfrac{\dfrac{\dfrac{\overline{A \wedge B \vdash A \wedge B}\ assume}{A \wedge B \vdash B}\ \wedge ER \qquad \dfrac{\overline{A \wedge B \vdash A \wedge B}\ assume}{A \wedge B \vdash A}\ \wedge EL}{A \wedge B \vdash B \wedge A}\ \wedge I}{\vdash (A \wedge B) \supset (B \wedge A)}\ \supset I$$

$$\dfrac{\dfrac{\overline{A \vee B \vdash A \vee B}\ assume \qquad \dfrac{\dfrac{\overline{A \vee B, A \vdash A}\ assume}{A \vee B, A \vdash B \vee A}\ \vee IR \qquad \dfrac{\overline{A \vee B, B \vdash B}\ assume}{A \vee B, B \vdash B \vee A}\ \vee IL}{A \vee B \vdash B \vee A}\ \vee E}{\vdash (A \vee B) \supset (B \vee A)}\ \supset I$$

4.5　古典論理と直観主義論理

前節で述べたのは命題論理の中でも「我々が日常使っている普通の論理」，すなわち古典論理 (classical logic) の体系である．古典命題論理の形式的体系は，真理値表による意味論とぴったり一致することが知られている．

定理 3（古典命題論理の健全性と完全性）　前節の体系で $\vdash A$ が導出できることと真理値表の意味論において A が恒真であること (つまり，どんな真理値割当てのもとでも真であること) は同値である．

従って，この観点に立てば，古典論理は最も自然な論理であると言える．しかし，古典論理における推論は必ずしも自然でないことがある．たとえば，排中律と呼ばれる命題 $A \vee \neg A$ を古典論理により導出してみよう．すなわち，判断 $\vdash A \vee \neg A$ を導出する．この判断には仮定はなく，結論の一番外側の論理記号は $\vee$ なので $\vee IL$ か $\vee IR$ 規則を最後に使って導出したと考えるのが自然である．しかし，実際にはどちらを使っても導出することはできない．(これは当然で，何の仮定もなく A や $\neg A$ が導出できては，どんな命題も導出できてしまう．)

実は，$\vdash A \vee \neg A$ を導くために最後に使う規則は，$\neg\neg E$ なのである．

$$\dfrac{\dfrac{\dfrac{}{\neg(A \vee \neg A) \vdash \neg(A \vee \neg A)}\ assume \qquad \begin{array}{c}\vdots\\ \neg(A \vee \neg A) \vdash A \vee \neg A\end{array}}{\dfrac{\neg(A \vee \neg A) \vdash \bot}{\vdash \neg\neg(A \vee \neg A)}\ \neg E}\ \neg E}{\vdash A \vee \neg A}\ \neg\neg E$$

このような導出により確かに排中律を導くことはできる．しかし， $A \vee \neg A$ より複雑な命題を多用するこのような導出は果たして自然なものであろうか？

古典論理では，背理法「$\neg A$ から矛盾を導ければ A と結論してよい」という推論法則も成立する．しかし，これも，意味から考えれば自然であっても，形式的体系においては，A を推論するためにそれより複雑な $\neg A$ を仮定する，という意味で自然な推論方法ではないだろう．これらの「不自然な推論法則」を列挙すると以下のものがあげられる．

- $\neg\neg E$ 規則
- 二重否定除去: $(\neg\neg A) \supset A$
- 排中律: $A \vee \neg A$
- 背理法:
$$\dfrac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}$$
- Peirce の法則: $((A \supset B) \supset A) \supset A$

実は，古典論理からこれらの法則を取り除いた論理が知られており，直観主義論理 (intuitionistic logic) と呼ばれる．言い換えれば，直観主義論理は，前に述べた古典論理の規則のうち $\neg\neg E$ 以外の 12 個から構成される論理体系である．

形式的体系としては，直観主義論理に基づいて，古典論理やそのほかの論理を見つめた方が，より自然であるので，本講義でもそのようにする．直観主義論理を基礎とした時，上記の 5 つの「不自然な」推論法則はすべて同値である．すなわち，直観主義論理に，上記の 5 つのうちのどれか 1 つを加えると古典論理が得られる．

4.6 直観主義論理と構成的解釈

前節で直観主義論理を導入したが，不整合な点を感じなかっただろうか？真理値表意味論と「ぴったり一致するもの」として古典論理の形式的体系を導入したのであるが，その古典論理の形式的体系に含まれる「不自然な推論」を排除して新たな体系を作ってしまった．この体系－直観主義論理の体系－は，古典論理の体系よりも弱いので，それに「ぴったり一致する」意味論は真理値表意味論ではないはずである．すなわち，以下の表の『?』を埋めるものは何であろうか？

論理	形式的体系	意味論
古典論理	$\neg\neg E$ 規則を含む 13 個	真理値表
直観主義論理	$\neg\neg E$ 規則以外の 12 個	?

ただし，真理値表の意味論も，直観主義論理の形式的体系に対応する意味論の 1 つである．なぜならば，直観主義論理の推論規則はすべて古典論理の推論規則でもあるので，直観主義論理で $\vdash A$ が導出できれば，それは古典論理でも導出できる．従って，命題 A は真理値表意味論で恒真式となり，健全性は成立している．

問題は，この逆が成立しないことである．真理値表意味論で恒真である命題の中には，直観主義論理では導出可能ではないものがある (たとえば，排中律)．つまり，完全性は成立していない．いま欲しいのは，健全性も完全性も成立する「ぴったり一致する意味論」である．

そのようなものの 1 つとして構成的解釈が知られている[*6]．ここでは，数学的に厳密な意味論を展開する余裕はないので，informal に構成的解釈を説明する．

定義 7 [構成的解釈 (命題論理に限定したもの)]

- 命題 A が正しい，とは，命題 A の証拠が (具体的な計算によって) 構成できることと定義する．証拠が構成できないときは命題 A は正しくない．
- 命題 $A \supset B$ の証拠とは，「命題 A の証拠をもらうと，命題 B の証拠を返す関数 (プログラム)」のこと．
- 命題 $A \wedge B$ の証拠とは，「命題 A の証拠と 命題 B の証拠の対 (ペア)」のこと．
- 命題 $A \vee B$ の証拠とは，「命題 A と命題 B のどちらか成立しているか」をあらわす 1bit の情報と，命題 A の証拠の対，もしくは，命題 B の証拠の対のこと．
- 命題 $\bot$ の証拠は存在しない．
- 命題 $\neg A$ の証拠とは，$A \supset \bot$ の証拠のこと．

なお，「p が $\neg A$ の証拠である」ことを言いかえると，「p が，A の証拠をもらって $\bot$ の証拠を返す関数」ということになるが，$\bot$ の証拠はないのだから，これは，「A の証拠が存在しない」ことと同値である．この場合，p はどんな関数でもよい．

上記の構成的解釈の定義は，「関数 (プログラム)」としてどのようなものが許されるかに依存しているが，ここでは，本講義の最初で述べた「計算可能関数」とする．つまり，Turing 機械，型のないラムダ計算，帰納的関数等のいずれかの方法で定義可能な関数のことである．

構成的解釈の例:

- $A \supset A$ の構成的解釈: その証拠は「A の証拠をもらって，A の証拠を返す関数」である．そのような関数は実際に $\lambda x.x$ として作ることができるので，$A \supset A$ は正しい．
- $(A \wedge B) \supset (B \wedge A)$ の構成的解釈: その証拠は「A の証拠と B の証拠の対をもらって，B の証拠と A の証拠の対を返す関数」である．そのような関数は実際にプログラムとして記述できる (対の左右をひっくり返すプログラム) ので，$(A \wedge B) \supset (B \wedge A)$ 正しい．
- $(\neg\neg A) \supset A$ の構成的解釈: まず，「p が $\neg\neg A$ の証拠である」ということを考えると，これは，「$\neg A$ の証拠が存在しない」ということである (p はどんな関数でもよい)．また，「q が $\neg A$ の証拠である」ということは，「A の証拠が存在しない」ということである (q はどんな関数でもよい)．これらから，「p が $\neg\neg A$ の証拠である」とは，「A の証拠が存在しないことはない」ということである．これは「A の証拠が存在する」ということである．[*7]ここで p はどんな関数でもよいので，p には何の情報もはいっていないことがわかる．

[*6] 話の都合上，このような順番で導入したが，歴史的には，構成的解釈や「直観主義」という哲学的な考え方が先にあり，それに対応する体系の 1 つとして直観主義論理体系が発生したのである．

[*7] 注意深い人は，ここの推論で，$\neg\neg E$ 規則を使っていることに気付くだろう．

最初にもどって $(\neg\neg A) \supset A$ が構成的解釈で正しいということは，「$(\neg\neg A)$ の証拠をもらって A の証拠を返す関数」が存在するということである．「$(\neg\neg A)$ の証拠」があるとしたら，上で一生懸命やった推論により，「A の証拠は存在する」ことがわかるが，その証拠がどういうものであるかは，さっぱりわからない．($\neg\neg A$ の証拠である p 自体には何の情報もはいっていなかったため，p を使ってもしょうがない．)

というわけで，$\neg\neg A$ の証拠をもらって A の証拠を返す関数というのは，作れそうもない．(A が正しいからといって，その証拠はいつでも自動的に生成できるとは限らない．) すなわち，$(\neg\neg A) \supset A$ は，構成的解釈のもとでは，正しくなさそうである．

- $A \vee \neg A$ の構成的解釈: その証拠は「A が正しいという情報」と「A の証拠」の対であるか，「「A が正しくないという情報」と「$\neg A$ の証拠」の対である．しかし，そのような情報をあらゆる A に対して作ることができるとは思えない．(命題 A だけを見て，A が正しいかどうかを判定するプログラムを書く必要がある．)

以上のような構成的解釈のもとで，$\neg\neg E$ 規則を除く 12 個の規則は全て，「正しい判断から正しい判断のみを導く」ということが証明できる．すなわち，直観主義論理の体系は，構成的解釈に関して健全性を満たす．また，$(\neg\neg A) \supset A$ や $A \vee \neg A$ などは，構成的解釈のもとで正しくなさそうであることがわかる．[*8]

構成的解釈は，実は，完全性も満たしている，ということが知られている．すなわち，本節の冒頭の表における 『?』 は構成的解釈である．構成的解釈は，後に，型付きラムダ計算と直感主義論理の対応を考える際にも重要な役割を果たす．

4.7 古典的な存在証明と構成的な存在証明

ここまで，古典命題論理と直観主義命題論理の差について，形式的体系の観点と，意味論の観点から議論してきた．ここでは，具体的な例題として有名な定理をあげる．

定理: 無理数 p と q で，p^q が有理数となるようなものが存在する．

この定理は以下のように証明できる．

証明: $R = \sqrt{2}^{\sqrt{2}}$ とおく．これは，有理数であるか無理数であるか，どちらである．そこで，場合分けする．

Case-1: R が有理数のとき，$p = \sqrt{2}, q = \sqrt{2}$ と置けば，定理の条件を満たしている．

Case-2: R が無理数のとき，$p = R, q = \sqrt{2}$ と置けば，$p^q = R^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} + \sqrt{2}^2 = 2$ となり，定理の条件を満たしている．(証明終わり)

この証明は数学の証明としては完璧である．しかし，この証明をいくら眺めても，具体的な p の値は計算できない．なぜなら，Case-1 と Case-2 のどちらになるかがわからないからである．たとえば，p が 1.5 より大きいか小さいかといったことすらわからない．せっかく証明を与えたのに，これでは，あまり役に立つとは言えない．すなわち，計算機科学の観点からすると，有意義な証明でない．

(補足: 上記の証明は、古典論理の証明であったが、2012 年度のこの授業の受講生である林さんは、直観主義論理での (構成的な) 証明を与えた。この証明では、$p = \sqrt{3}, q = \log_3 4$ と置くと、$p^q = 2$ となる、というものである。p と q が無理数であることの証明は難しくない。)

上記の「有意義でない」ことの原因を分析する．上の証明は，最初に「(R が有理数)$\vee$(R が無理数)」とい

[*8] ここで言えるのは，「正しくないと思われる」というだけであり，本当に正しくないことを示すためには，「プログラムによって計算できる関数とはなにか」ということを知らなければならない．ここでは計算可能関数の議論に深入りしない．

うことを使った．これは，「(R が有理数)$\vee$ $\neg$(R が有理数)」と言いかえるとわかるように排中律 $A \vee \neg A$ の一種である．すなわち，上の証明は，直観主義論理では成立せず，古典論理における証明となっている．

実は，以下のことが成立する．

- 古典論理の方が，直観主義論理よりも証明できるものは多い．($A \vee \neg A$ や $\neg\neg A \supset A$ などが証明できる．)
- 直観主義論理で証明できたものは，構成的解釈を満たすので，計算にとって有用な情報 (存在定理の場合は，その存在するものを具体的に計算する情報) を含んでいる．

従って，もし，上記の定理を直観主義論理で証明していれば，「... となる p と q が存在する」という形の定理からは，必ず，p と q を具体的に計算する関数 (プログラム) が得られたはずである．面倒だからといって手を抜いて $A \vee \neg A$ を使ってしまったので，情報量の少ない古典論理の証明しか得られなかった，ということになる．

このように，「何かを具体的に求めたい」と思うなら，直観主義論理を使うべきである．一方，「具体的な計算は良いので，正しいかどうかだけを知りたい」と思うなら，古典論理を使った方が簡単である．

4.8 導出の簡約 (計算)

ラムダ計算などの計算体系は「計算」という本質的に動的なものを記述する体系である．では，論理体系に動的な概念はあるだろうか？その答えは YES であり，導出自身を変形していく計算について考える．

まず，導出の無駄という概念を考えよう．

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ \vdots & \vdots \\ \Gamma \vdash A & \Gamma \vdash B\end{array}}{\Gamma \vdash A \wedge B}\wedge I}{\Gamma \vdash A}\wedge EL$$

このような導出は，その部分導出である $\mathcal{D}$ の部分で既に $\Gamma \vdash A$ という結論が出ているので，最後の 2 回の推論は無駄である．すなわち，上記の導出は，結論となる判断を保ったまま，以下の (より簡単な) 導出に変形することができる．

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ \vdots & \vdots \\ \Gamma \vdash A & \Gamma \vdash B\end{array}}{\Gamma \vdash A \wedge B}\wedge I}{\Gamma \vdash A}\wedge EL \quad \rightsquigarrow \quad \begin{array}{c}\mathcal{D} \\ \vdots \\ \Gamma \vdash A\end{array}$$

これは，$\wedge I$ の直後に $\wedge EL$ を適用した，という形を除去したという変形である．同様に，$\wedge I$ の直後に $\wedge ER$ を適用した，という形を除去する変形が考えられる．

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D} \\ \vdots \\ \Gamma, A \vdash B\end{array}}{\Gamma \vdash A \supset B}\supset I \quad \begin{array}{c}\mathcal{E} \\ \vdots \\ \Gamma \vdash A\end{array}}{\Gamma \vdash B}\supset E$$

この導出は，$\mathcal{D}$ の部分で B を結論とする導出ができているので，それに変形できそうである．しかし，いきなり $\mathcal{D}$ に置きかえてしまっては，Γ だった仮定が Γ, A に変わってしまい，(仮定が増えるということは，全体としては主張が弱くなるので) まずい．しかし，A の証明は既に $\mathcal{E}$ の部分でできるので，これを張り合わせればよさそうである．張り合わせる先は $\mathcal{D}$ の中で仮定 A を実際に使っているところ，すなわち $\Gamma, A, \Delta \vdash A$ の形の assume 規則である．ここでは，assume 規則が 2 回使われているとした場合の例をあげる．

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A, \Delta_1 \vdash A}\, as \quad \dfrac{}{\Gamma, A, \Delta_2 \vdash A}\, as}{\vdots\, \mathcal{D}}}{\Gamma, A \vdash B}}{\Gamma \vdash A \supset B}\supset I \qquad \dfrac{\vdots\, \mathcal{E}}{\Gamma \vdash A}}{\Gamma \vdash B}\supset E \quad \rightsquigarrow \quad \dfrac{\vdots\, \mathcal{E} \qquad \vdots\, \mathcal{E}}{\Gamma, \Delta_1 \vdash A \qquad \Gamma, \Delta_2 \vdash A}\\ \vdots\, \mathcal{D}\\ \Gamma \vdash B$$

しかしこれでもまだ問題がある．$\mathcal{E}$ はもともと $\Gamma \vdash A$ を結論とする導出だったのに，変形後は $\Gamma, \Delta_i \vdash A$ 等を結論とする導出になってしまっていて，うまく張り合わせることができない．また，$\mathcal{D}$ の仮定も少し違っている．そこで，$\mathcal{D}$ と $\mathcal{E}$ に以下の変形をしたものを $\mathcal{D}'$ と ${\mathcal{E}'}_i$ とする．

- $\mathcal{D}$ に含まれる仮定列から A を取り除いたものを $\mathcal{D}'$ とする．
- $\mathcal{E}$ に含まれる仮定列に Δ_i を付け加えたものを ${\mathcal{E}'}_i$ とする．

このようにして以下の変形を得る．

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A, \Delta_1 \vdash A}\, as \quad \dfrac{}{\Gamma, A, \Delta_2 \vdash A}\, as}{\vdots\, \mathcal{D}}}{\Gamma, A \vdash B}}{\Gamma \vdash A \supset B}\supset I \qquad \dfrac{\vdots\, \mathcal{E}}{\Gamma \vdash A}}{\Gamma \vdash B}\supset E \quad \rightsquigarrow \quad \dfrac{\vdots\, {\mathcal{E}'}_1 \qquad \vdots\, {\mathcal{E}'}_2}{\Gamma, \Delta_1 \vdash A \qquad \Gamma, \Delta_2 \vdash A}\\ \vdots\, \mathcal{D}'\\ \Gamma \vdash B$$

この変形では，$\mathcal{E}$ を 2 つコピーしているので，必ずしも導出の大きさを小さくする変形ではないが，$\supset I$-$\supset E$ 規則が連続する部分の無駄を省く効果はある．

次は，$\vee$ の場合である．

$$\dfrac{\dfrac{\vdots\, \mathcal{D}}{\Gamma \vdash A}}{\Gamma \vdash A \vee B}\vee IL \qquad \dfrac{\dfrac{}{\Gamma, A, \Delta_i}\, as}{\vdots\, \mathcal{E}} \qquad \dfrac{\vdots\, \mathcal{F}}{\Gamma, B \vdash C}}{\Gamma \vdash C}\vee E \quad \rightsquigarrow \quad \dfrac{\vdots\, {\mathcal{D}'}_i}{\Gamma, \Delta_i}\\ \vdots\, \mathcal{E}'\\ \Gamma \vdash C$$

この変形においても，$\mathcal{E}$ の中で仮定 A を使う assume ルールを $i = 1, 2, \cdots$ して，$\mathcal{E}'$ は $\mathcal{E}$ に含まれる仮定列から A を取り除いた導出，${\mathcal{D}'}_i$ は $\mathcal{D}$ に含まれる仮定列に Δ_i を付け加えた導出とする．

同様に，$\vee IR$-$\vee E$ の変形も定義できる．また，$\neg A$ は $A \supset \bot$ の省略形と見なすことにより，上記変形操作を適用することができる．

一般に，$\supset, \wedge, \vee, \neg$ という論理記号に対する導入規則と除去規則が連続して現れている部分は，導出の中での「無駄」である．無駄のある導出に対して，上記のような操作によって無駄を除去する操作を導出に対する「計算」(簡約) と定めることができる．

ところで，通常人間が書くのは，無駄のない導出のみなので，上記の変形操作にどんな意味があるのか，こんなものを計算と呼んだところで極めて特殊な計算に過ぎない，と疑問に思うかもしれない．実はそうでない．

定理 4（直観主義命題論理の導出の変形） 上記の変形操作を「計算」と見なすと，合流性，(強い) 停止性を満たす．

この定理自身，驚くべき内容である．というのは，$\supset$ や $\vee$ に関する変形操作で見たように，変形後の導出は変形前より小さくなる保証が全くないからである．(コピーをいくつも作成するので，しばしば，非常に大きくする効果をもつ)．

しかし，ここではこの定理の証明は置いておき，その意味を考えてみよう．この定理から，直観主義命題論理では，導出可能な判断に対して，無駄のない導出が必ず存在することがわかる．このことは，命題の導出において大きな意味を持っている．以下の導出が無駄のない導出であったとしよう．

$$\dfrac{\dfrac{\overline{J_{3-1}} \quad \overline{J_{3-2}}}{J_{2-1}}\, R_2 \qquad \dfrac{\overline{J_{3-3}} \quad \overline{J_{3-4}}}{J_{2-2}}}{J_1}\, R_1$$

この導出のうち，結論の判断 J_1 から，一番左の道を下からたどってみよう．そこにあらわれる，判断と推論規則名の列を $J_1, R_1, J_{2-1}, R_2, J_{3-1}, R_3, J_{4-1}, \cdots, R_n$ とする．この系列では，R_k が除去規則 (規則名に E がつく規則) で，R_{k+1} が導入規則 (規則名に I がつく規則) であるような状況が起きないことを示す．そのような状況が起きたとすると，R_k は，J_{k+1} の結論となる命題の主たる論理記号 (一番外側の論理記号) に関する除去規則であり，R_{k+1} は，その同じ論理記号に関する導入規則である．すなわち，この部分は「無駄」になっているので変形可能であるが，上記導出は無駄がないと仮定したので矛盾である．よって，上のような状況は起きない．すなわち，$R_1, R_2, R_3, \cdots$ という規則名の系列を取りだすと，$R_1, \cdots, R_m$ が導入規則，$R_{m+1}, \cdots, R_{n-1}$ が除去規則，R_n が assume 規則となっているはずである．ただし，$m = 0$ (導入規則が一度も使われない) や，$m = n - 1$ (除去規則が一度も使われない) ということもある．

この解析により，何らかの判断を導出する場合，主たる道筋では (つまり，導出の一番左側の道では)

- 導入規則を何回か使って結論をばらばらにする．
- 次に除去規則を何回か使って仮定に到達する．

という道筋だけを検討すればよいことがわかる．そのような道筋で導出できれば良いし，導出できなければ，他の道筋を検討することなく，「導出不能」ということがわかる[*9]．

これは定理の自動証明をする上で極めて重要な知見である．たとえば，$\vdash A \supset B$ という形の判断を導出するという問題が与えられたとすると (つまり，それが導出可能であれば)，その最終規則は必ず $\supset I$ である導

[*9] そんな重要な情報は演習をやる前に教えてほしかった，と思う人がいるかもしれないが，学問とはそんなものである．簡単にできる方法を最初に学んでしまうと，その方法がいかに偉大なものかがわからない．

出が存在する．そのほかの導出もあるかもしれないが，考えなくてよい，ということである．

このことを精密化させると以下の定理が成立する[*10]．

定理 5（(直観主義命題論理の導出における) 部分命題の性質） $\Gamma \vdash A$ が導出可能なとき，Γ と A の部分命題のみから構成される導出が存在する．

命題 A の部分命題とは，A を構成する途中に出現する命題のことであり，たとえば，$A \supset (B \wedge C)$ の部分命題は，$A, B, C, B \wedge C, A \supset (B \wedge C)$ の 5 つである．上記の定理は，一般には「部分論理式の性質」とよばれる定理であり，何かの命題を証明したいときには，その命題の部分論理式のみを対象に導出を構成することを考えればよい，というものである．この定理が成立するとき，定理証明の探索の手間が劇的に減ることが想像できよう．直観主義論理は部分論理式の性質が成立するという意味で非常にたちのよい体系である．

一方，古典論理では，上記のような良い性質は成立しない．実際，$\vdash A \vee \neg A$ に対しては，いきなり除去規則 $\neg\neg E$ を適用して，結論を $\vdash \neg\neg(A \vee \neg A)$ にしてから導入規則を使うような「不自然な」導出しか存在しないことを既に見てきた．この場合，$\neg\neg(A \vee \neg A)$ は，$A \vee \neg A$ の部分命題ではないので，上記定理の反例となっている．

付記: 導出に対する変形操作 (仮定列を増やしたり減らしたり) を厳密に述べるのはかなり面倒なので，本節では極めて informal に述べた．詳細は，論理学の本格的な教科書を参照されたい．ただし，次章で，導出に対する変形操作と本質的に同じ操作について説明する．

[*10] 本節で述べた「変形」の範囲では，一番左側の道についてしか上のような良い性質は言えない．以下の定理を証明するためには，変形操作をさらに増やして精密化させる必要があるが，ここでは省略する．

5　型付きラムダ計算

プログラム言語は、コンピュータ・プログラムを記述する言葉であり，良いソフトウェアの構築には，良いプログラム言語の利用が欠かせない．現代のプログラム言語に欠かせない要素が、型システムである。本講義では、型システムを中心に置いて、プログラム言語の理論的基礎を学ぶ。

5.1　型の概念

型付きラムダ計算は，型のないラムダ計算に型 (Type) の概念を導入した計算体系である．では，型とは何だろうか？

C, Java, Fortran, ML など、多くのプログラム言語は，型システムを持っている．型には，整数型や浮動小数点型などの基本的なものから，配列型，レコード型，構造体の型、関数の型など複合的なものまである。プログラム実行時に整数と配列を加えようとしたり、整数型の変数に構造体を代入しようとするのはエラーであり、このようなエラーを実行時に出すのではなく、静的に (つまり、実行するより前に) 出すようにするのが型システムの役割である[*11]。

型は、より正確にいうと，同じ操作が可能な一群のデータを特徴付ける情報であり，個々のデータの値によって変わらない．従って，実行時に変数の型が変化することは通常はないため，実行前 (コンパイル時) に，型の整合性を判定することができる．すなわち，動的に思われる「計算」の世界における**静的な概念の代表選手**が，型である．

現代的プログラム言語の多くが型システムを持っているのは，型の概念が有用であることを示している．実際，プログラミング上の些細な (しかし，頻繁に起きる) 誤りの多くは型エラーの形で発見することができる．また，場合によっては，型情報を生かして実行速度の向上を図ることができる．

型の整合性の判定は，簡単に思えるかもしえない。たとえば、「int 型の変数に char 型の値を代入しようとした」というエラー判定は容易である。しかし、このような基本的な型だけでなく、型構成子がある場合，必ずしも簡単ではない。

例 9

$$S = \lambda x.\lambda y.\lambda z.\ (x\ z)\ (y\ z)$$

$$K = \lambda x.\lambda y.\ x$$

とするとき， S や K はどういう型をもつ関数か？

この講義では，型のないラムダ計算に型システムを導入した型付きラムダ計算の体系をとりあげる．ここで扱うのは単純型付きラムダ計算と呼ばれる最もシンプルな体系をやや拡張したものである．

5.2　構文

プログラムの構文を定義する前に，型の構文を定義する．

[*11] ここでは、「型システム」という言葉を「静的な型システム」に限定した言葉遣いをしている。Ruby や Perl など、静的な型の整合性の検査をしないプログラム言語でも、「型」の概念を持っているものがあり、実行時に型の整合性を検査することがある。このような言語を、動的型付け言語と言う。

ここでは，int, String など基本となる型の内部構造には立ち入らないので，それらを単に $K_1, K_2, \cdots, K_n$ という型定数で表示する．型定数の中には、特別な型定数 $\bot$ が含まれるとする。これは、「空の型」を意味する。

$$\frac{}{K_i : Type} \qquad \frac{A : Type \quad B : Type}{A \times B : Type} \qquad \frac{A : Type \quad B : Type}{A + B : Type} \qquad \frac{A : Type \quad B : Type}{A \to B : Type}$$

$\times$ は直積, $+$ は直和, $\to$ は関数空間を表す型である．それらの意味は，後に述べる計算規則のところで明らかになる．

次に，項の構文を定義する．この際以下の 2 つの点に注意する．

- 型付きラムダ計算においてはすべての項は型を (整合的に) 持たなけれければいけないので，単に「M が項である」ということを推論する規則ではなく，「M が型 A の項である」ということを推論する規則とする．
- さらに，項の構成の過程で，M に含まれる変数がどういう型を持つかを覚えておく必要がある．そのことを， $x_1 : A_1, \cdots, x_n : A_n$ という形の列として表現することにして，一般にこういう列を Γ という文字であらわす．この列のことを「宣言」と呼ぶ.

以上の 2 点から，項の構文の構成規則は $\Gamma \vdash M : A$ という形を推論するための規則となることがわかる．ここで $\vdash$ や : という記号には深い意味はなく，(Γ, M, A) という 3 つ組の推論規則でもよかったのだが，伝統的な記法に従った．

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \texttt{var}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B}\ \texttt{pair} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \texttt{left}(M) : A}\ \texttt{left} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \texttt{right}(M) : B}\ \texttt{right}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{inl}(M) : A + B}\ \texttt{inl} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \texttt{inr}(M) : A + B}\ \texttt{inr}$$

$$\frac{\Gamma \vdash M : A + B \quad \Gamma, x : A \vdash N : C \quad \Gamma, y : B \vdash L : C}{\Gamma \vdash \texttt{case}(M, (x : A)N, (y : B)L) : C}\ \texttt{case}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda(x : A)M : A \to B}\ \texttt{lambda} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M@N : B}\ \texttt{apply}$$

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \texttt{abort}(M) : A}\ \texttt{abort}$$

それぞれの規則の横棒の右隣に規則名を書いた．(var, pairなど)

型なしラムダ計算における規則に対応するのは，var, lambda, applyの 3 つであり，それ以外は，ここで新しくでてきたものである．$\lambda(x : A)M$ は，型なしラムダ計算における $\lambda x.M$ という項に x の型 A の情報を付加したものである．この記法には，いろいろな種類があり，$\lambda x : A.M$ と書いたり，$\lambda x^A.M$ と書くことがある．ここでは，演習システムの整合性の観点から，$\lambda(x : A)M$ という記法を採用した．(式の表記では，不要なかっこを省略できるのが普通だが，このラムダ式のかっこは，省略しないものとする．)

$M@N$ という形の項は、型なしラムダ計算における $M\ N$ と同様に，関数 M に引数 N を適用させた式，すなわち，関数適用 (function application) をあらわす．数学ならば，$M(N)$ というように，引数に括弧をつ

けるが，ラムダ計算では，引数の括弧は (曖昧さが生じない限り) 省略する．そのかわりに，ここでは，単に $M\ N$ と並べるのではなく (並べる表記は，ラムダ計算に成れていない人には間違いやすい)，$M@N$ というように，@ の記号をいれて，関数適用を明記するようにした．このテキストでは，ときどき，@を省略した記法を使うことがあるが，演習システムでは @ は省略できないことに注意されたい．

pairは直積型の項を作る規則であり，left, rightは直積型の項を使う規則である．inl, inrは直和型の項を作る規則であり，caseは直和型の項を使う規則である．これらの意味は必ずしも直感的に想像できないかもしれないが，後で計算規則がでてきたときに明らかになる．ここでは，項は上の規則によって，機械的に (コンピュータでもわかるように) 構成される，ということを理解してほしい．

また，宣言 Γ が必ずしも一定でないことにも注意してほしい．たとえば，$\lambda(x:A)M$ という項は，x という変数が束縛されるので，M に対する宣言である $\Gamma, x:A$ より $x:A$ が減り，Γ だけになっている．(M の中で x が使われていなくてもよい．) 同様に，$\mathtt{case}(M,\ (x:A)N,\ (y:B)L)$ という項では，N の中に x, L の中に y が自由に現れていてもよいのだが，それらは，$\mathtt{case}(M,\ (x:A)N,\ (y:B)L)$ という項の中では束縛されるということを表している．

上記の規則を有限回適用して $\Gamma\ \vdash\ M:A$ が推論できたときに「宣言 Γ のもとで M は型 A を持つ項である」という．(Γ が空の列のとき，「M は型 A を持つ項である」ということもある．)

例 10　型つきラムダ計算の項の構成 (型付け) の例をあげる．

$$\dfrac{\dfrac{}{x:A\ \vdash\ x:A}\ \mathtt{var}}{\vdash\ \lambda(x:A)x:A\to A}\ \mathtt{lambda}$$

$$\dfrac{\dfrac{\dfrac{}{x:A\times B\ \vdash\ x:A\times B}\ \mathtt{var}}{x:A\times B\ \vdash\ \mathtt{right}(x):B}\ \mathtt{right} \quad \dfrac{\dfrac{}{x:A\times B\ \vdash\ x:A\times B}\ \mathtt{var}}{x:A\times B\ \vdash\ \mathtt{left}(x):A}\ \mathtt{left}}{x:A\times B\ \vdash\ (\mathtt{right}(x),\mathtt{left}(x)):B\times A}\ \mathtt{pair}}{\vdash\ \lambda(x:A\times B)(\mathtt{right}(x),\mathtt{left}(x)):(A\times B)\to(B\times A)}\ \mathtt{lambda}$$

$$\dfrac{\dfrac{\dfrac{}{x:A+B\ \vdash\ x:A+B}\ \mathtt{var} \quad \dfrac{\dfrac{}{x:A+B,y:A\ \vdash\ y:A}\ \mathtt{var}}{x:A+B,y:A\ \vdash\ \mathtt{inr}(y):B+A}\ \mathtt{inr} \quad \dfrac{\dfrac{}{x:A+B,z:B\ \vdash\ z:B}\ \mathtt{var}}{x:A+B,z:B\ \vdash\ \mathtt{inl}(z):B+A}\ \mathtt{inl}}{x:A+B\ \vdash\ \mathtt{case}(x,(y:A)\mathtt{inr}(y),(z:B)\mathtt{inl}(z)):B+A}\ \mathtt{case}}{\vdash\ \lambda(x:A+B)\mathtt{case}(x,(y:A)\mathtt{inr}(y),(z:B)\mathtt{inl}(z)):(A+B)\to(B+A)}\ \mathtt{lambda}$$

$$\dfrac{\dfrac{\dfrac{}{x:\bot\ \vdash\ x:\bot}\ \mathtt{var}}{x:\bot\ \vdash\ \mathtt{abort}(x):\bot\to A}\ \mathtt{abort} \quad \dfrac{}{x:\bot\ \vdash\ x:\bot}\ \mathtt{var}}{x:\bot\ \vdash\ \mathtt{abort}(x)@x:A}\ \mathtt{apply}$$

5.3　型検査と型推論

判定問題とは，入力に対して何らかの性質が成立するかどうか、つまり、YES か NO かを判定する問題である．判定問題をプログラムによって有限時間内に必ず解くことができるとき，決定可能 (decidable) という．

型のある計算体系に対して，以下の 3 つの判定問題を解くことが重要である．

- Γ, M, A が与えられたとき，$\Gamma \vdash M : A$ が導けるか？ (type checking, 型検査問題)
- M が与えられたとき，$\Gamma \vdash M : A$ が導ける Γ, A があるか？ (type inference, 型推論問題その 1)
- Γ, M が与えられたとき，$\Gamma \vdash M : A$ が導ける A があるか？ (type inference, 型推論問題その 2)
- Γ, A が与えられたとき，$\Gamma \vdash M : A$ が導ける M があるか？ (inhabitance)

これらの問題が決定可能であるかどうかは，型システムに依存して決まるが，型検査より型推論の方が難しいことが多い．

型システムを持つほとんど全てのプログラム言語において，型検査問題あるいは型推論問題は決定可能であり，言語処理系であるコンパイラに組み込まれている．(C 言語等では，Γ, A が完全に与えられて整合性の判定をしているので，型検査問題である．一方，ML 言語等では，型の情報はプログラム中に与えられておらず，コンパイラが型推論しなければならない．また，JAVA バイトコードの verifier なども型情報がないところから型情報を復元しているので，型推論問題を解いていることになる．)

本講義の型つきラムダ計算の体系に対しては，以下の定理が容易に示せる．

定理 6 先に与えた型付きラムダ計算の体系に対して，型検査問題，型推論問題は決定可能である．

ここではこの定理の証明 (型検査および型推論アルゴリズム) を与えるかわりに，演習によって，実際にどのようなアルゴリズムで型推論できるかを体験してもらうことにする．

5.4 Church 流と Curry 流

本講義で扱っている型つきラムダ計算は，Church style と呼ばれるものである．これは，ラムダ式の構成において，$\lambda(x : A)M$ のように必ず型 A を記述する流儀である．このため，型検査や型推論が簡単になっている．C 言語などで，ブロックごとに局所変数の型を宣言するのは，Church style と同じである．

```
int foo (int x)
{
   int y;
   y = x * 2 + 1;
   return y + 3;
}
```

一方，ラムダ式の構成において，$\lambda x.M$ のように型 A を記述しない流儀もあり，Curry style と呼ばれる．こちらでは，型検査や型推論を行う際に，x の型を推論しなければいけないので，問題が難しくなる．これは，ML のように変数の型を宣言しない言語に対応している．

```
let foo x =
  let y = x * 2 +1 in
    y + 3
```

型検査および型推論については、後の章で詳しく学習する。

5.5 計算規則

型付きラムダ計算においても，型なしラムダ計算と同じ β-簡約規則が計算規則の中心である．しかし，それ以外に，直積型や直和型を導入したので，それらに対応する計算規則も導入される．計算規則を定義するための準備として、代入の定義をおこなう。

定義 8 [代入の定義] x と N は同じ型を持つとする。項 M の中の自由な x の出現に 項 N を代入して得られる項 $M\{x := N\}$ は以下のように定義される項である．(ここで $\stackrel{\text{def}}{=}$ は、定義をあらわす等号である。あとで、「計算すると等しい」ということを意味する等号もでてくるので、ここでは区別している。)

- $x\{x := N\} \stackrel{\text{def}}{=} N$
- $y\{x := N\} \stackrel{\text{def}}{=} y$, ただし、$x$ と y が異なる変数
- $(L@M)\{x := N\} \stackrel{\text{def}}{=} (L\{x := N\})@(M\{x := N\})$
- $(\lambda(y : A)M)\{x := N\} \stackrel{\text{def}}{=} \lambda(y : A)(M\{x := N\})$, ただし、$y$ は x と異なり、N に自由な出現を持たない変数
- $f(L)\{x := N\} \stackrel{\text{def}}{=} f(L\{x := N\})$, ただし、$f$ は 1 引数の関数記号 ($\texttt{left}$, $\texttt{right}$, $\texttt{inl}$, $\texttt{inr}$, $\texttt{abort}$)
- $g(L, M)\{x := N\} \stackrel{\text{def}}{=} g(L\{x := N\}, M\{x := N\})$, ただし、$g$ は 2 引数の関数記号 $((\_, \_))$
- $\texttt{case}(M, (y : A)L, (z : B)P)\{x := N\} \stackrel{\text{def}}{=} \texttt{case}(M\{x := N\}, (y : A)(L\{x := N\}), (z : B)(P\{x := N\}))$, ただし、$y, z$ は x と異なり、N に自由な出現を持たない変数

この定義は、型のないラムダ計算のときと同様のものを、型付きラムダ計算の項に適用したものであるが、それだけでなく、煩雑さを避けるため変更した箇所がある。

それは、4 行目の $(\lambda(y : A)M)\{x := N\}$ の定義である。ここでは、「y は x と異なり、N に自由な出現を持たない変数」という条件を付けている。それを満たした場合には、通常の代入の定義と同じである。しかし、この条件は必ず満たされるわけではなく、満たされない場合は、上記の定義では代入ができないことになる。代入という操作そのものは、どんな項 N, M とどんな変数 x に対しても定義できてほしいので、これは欠陥であるが、この問題の解決は後の章でおこなうことにして、ともかく、上記の条件を満たしているときのみ代入できるとしておこう。$\texttt{case}$ の形の項に対する定義も同様である。(一般に、変数の束縛を伴う項は同様である。)

次に、(項に関する) 文脈を定義する。文脈は、変数を 1 つ以上持つ項において、変数を 1 か所だけ、穴 (ホール) とよばれる □ で置きかえたものである。ただし、この節では型付きラムダ計算であるので、正確には「型付けられた項において、変数を 1 か所だけ、□ で置きかえたもの」ということになる。

例 11 文脈の例: $(\lambda(x : A + B)\texttt{case}(x, (y : A)\texttt{inr}(y), (z : B)\texttt{inl}(\square)))@(\texttt{inl}(a))$

このように、変数がでてきてよい場所のうち 1 か所だけを □ にしたものである。

文脈に対して、穴埋め (hole-filling) という操作が定義できる。具体的には、文脈 C における穴を (型がつく) 項 N で置きかえて得られる項を、$C[N]$ と書き、この操作を穴埋めという。

例 12 文脈の穴埋めの例: 上記の文脈を C とし、N を (x, z) とすると、$C[N]$ は $(\lambda(x : A + B)\texttt{case}(x, (y : A)\texttt{inr}(y), (z : B)\texttt{inl}(x, z)))@(\texttt{inl}(a))$ である。

文脈の穴埋めは、代入とよく似ているが、決定的に異なるのは、代入と違って、変数の束縛関係を無視して、ただその場所に書きこんでしまうことである。実際、上記の例でも、N は変数 z を含んでおり、これは文脈 C の穴においては束縛変数であるが、そんなことは構わず、N をのままいれてしまうのが「穴埋め」操作である。

型付きラムダ計算における計算規則は次のように与えられる.

定義 9 [計算規則]

$$\begin{aligned}
(\lambda(x:A)M)@N &\to M\{x := N\} \\
\texttt{left}(M,N) &\to M \\
\texttt{right}(M,N) &\to N \\
\texttt{case}(\texttt{inl}(M),(x:A)N,(y:B)L) &\to N\{x := M\} \\
\texttt{case}(\texttt{inr}(M),(x:A)N,(y:B)L) &\to L\{y := M\} \\
C[M] &\to C[N] \quad (M \to N \text{ のとき})
\end{aligned}$$

最初の規則は型なしラムダ計算と同じ β-規則であり，関数型 $A \to B$ を持つ項を計算する規則である.

2-3 番目の規則は直積の型 $A \times B$ を持つ項を計算する規則である．直積型を持つ項，(M,N) の左側の要素を取れば M になるはずであり，右側の要素を取れば N になるはずである，ということを表している.

4-5 番目の規則は直和の型 $A + B$ を持つ項を計算する規則である．case は場合分けを行うものであり，意味的には，以下の if-then-else 文と似ている.

```
if (M) {
   L
} else {
   N
}
```

ここで M は 1 か 0 のいずれかの値を取る式とすると，M が 1 のとき L が実行され，M が 0 のとき N が実行される.

$\texttt{case}(M,(x:A)L,(y:B)N)$ という項の場合，if-then-else を少し拡張していて，M は 1 か 0 のいずれかの値を取る式ではなく，$\texttt{inl}(P)$ の形か $\texttt{inr}(Q)$ の形のいずれかである．場合分けは，M が inl の形か inr の形かに従って行われ，それぞれの場合において，P や Q の値を使う．つまり，以下のような実行をするのである.

```
if (M が inl(P) の形のとき) {
   x = P;
   L
} else if (M が inr(Q) の形のとき) {
   y = Q;
   N
}
```

ここまで来れば，なぜ，$\texttt{case}(M,(x:A)L,(y:B)N)$ という項において L の中の x や N の中の y の出現が束縛されている，と約束したかがわかるであろう.

計算規則の中の 6 番目の規則は、大きな項の内部で計算可能な部分があったときに計算を進める規則である。ここで $C[M]$ は、大きな項を表し、そのうち (この) 計算に関係ある部分だけを抽出して M と書いた。M 以外の部分は (この) 計算に関係ないので、C という文脈の形であらわす。M を計算して N になったとき、M 以外の部分は変わらないので、得られる項は $C[N]$ である。

例 13　型付きラムダ計算における計算の例をあげる.

$$
\begin{aligned}
&(\lambda(x : A \times B)(\mathtt{right}(x), \mathtt{left}(x)))@((a, b)) \\
\to\ &(\mathtt{right}(x), \mathtt{left}(x))\{x := (a, b)\} \\
\equiv\ &(\mathtt{right}(a, b), \mathtt{left}(a, b)) \\
\to\ &(b, \mathtt{left}((a, b))) \\
\to\ &(b, a)
\end{aligned}
$$

$$
\begin{aligned}
&(\lambda(f : A \to B)\lambda(x : A)f@x)(\lambda(y : A)b)@a \\
\to\ &((\lambda(x : A)f@x)\{f := \lambda(y : A)b\})@a \\
\equiv\ &(\lambda(x : A)(\lambda(y : A)b)@x)@a \\
\to\ &((\lambda(y : A)b)@x)\{x := a\} \\
\equiv\ &(\lambda(y : A)b)@a \\
\to\ &b\{y := a\} \\
\equiv\ &b
\end{aligned}
$$

$$
\begin{aligned}
&(\lambda(x : A + B)\mathtt{case}(x, (y : A)\mathtt{inr}(y), (z : B)\mathtt{inl}(z)))@(\mathtt{inl}(a)) \\
\to\ &\mathtt{inr}(y)\{y := a\} \\
\equiv\ &\mathtt{inr}(a)
\end{aligned}
$$

$$
\begin{aligned}
&(\lambda(x : A + B)\mathtt{case}(x, (y : A)\mathtt{inr}(y), (z : B)\mathtt{inl}(z)))@(\mathtt{inr}(b)) \\
\to\ &\mathtt{inl}(z)\{z := b\} \\
\equiv\ &\mathtt{inl}(b)
\end{aligned}
$$

計算が進んでも，項の型が変わらないこと，計算そのものは型情報と関係なく進んでいることに注意してほしい．言い換えれば，「計算」という動的な概念に対して，「型」は静的な概念である．

5.6　α 同値と代入

前の節で述べた代入の定義は、「すべての項 M, N と変数 x に対して $M\{x := N\}$ が定義できる」が成立しない、という点で不満足なものである。たとえば、$(\lambda(x : A)y)\{y := x\}$ という代入は、条件を満たさず定義できない。これでは、計算が途中でつっかえてしまうことがあり不都合である。

この定義の欠陥を補うため、通常は、以下の概念を導入する。

> α 同値性: 変数 y が項 M にあらわれない変数とする。このとき項 M と項 $M\{x := y\}$ を「同じ」と見なす。

これによって、代入をする前にあらかじめ、束縛変数の名前を変更しておいて、変数名の衝突を避けることができる。

このための手段として、多くの教科書では α 同値に基づいた定義をしている。たとえば、述語論理において $\forall x.A(x)$ と $\forall y.A(y)$ は同じであるとか、積分において $\int_0^\infty f(x)dx$ と $\int_0^\infty f(y)dy$ は同じ、といったことを習ったことがある人も多いだろう。このような立場を取る根拠は、束縛変数の名前だけを一斉に変更しても、その式が表現している「もの」は同じである、ということによる。

α 同値性は、直感的には (人間には) 簡単な概念であるが、数学的に正確な定義はやや厄介である。この節では一応きちんと導入しておくが、この授業の目標としては、「代入」がきちんとできれば問題ない。(なお、演習で使うシステムは、α 同値性の計算を勝手にやってくれる。)

α 同値性の定義の前に、代入の定義においてはきちんと説明せずに使っていた「変数 y が、項 N に自由に出現を持つ」ことの定義をしておこう。

定義 10 [自由変数] 項 M の自由変数の集合 $\mathtt{FV}(M)$ を、次のように定義する。

- $\mathtt{FV}(x) \stackrel{\text{def}}{=} \{x\}$
- $\mathtt{FV}(L@M) \stackrel{\text{def}}{=} \mathtt{FV}(L) \cup \mathtt{FV}(M)$
- $\mathtt{FV}(\lambda(y:A)M) \stackrel{\text{def}}{=} \mathtt{FV}(M) - \{y\}$
- $\mathtt{FV}(f(L)) \stackrel{\text{def}}{=} \mathtt{FV}(L)$、ただし、$f$ は 1 引数の関数記号 (`left`, `right`, `inl`, `inr`, `abort`)
- $\mathtt{FV}(g(L,M)) \stackrel{\text{def}}{=} \mathtt{FV}(L) \cup \mathtt{FV}(M)$、ただし、$g$ は 2 引数の関数記号 $((\_,\_))$
- $\mathtt{FV}(\mathtt{case}(M,(y:A)L,(z:B)P)) \stackrel{\text{def}}{=} \mathtt{FV}(M) \cup (\mathtt{FV}(L) - \{y\}) \cup (\mathtt{FV}(P) - \{z\})$

ここで、集合 S,T に対して $S-T$ は集合の差集合、すなわち、S から $S \cap T$ を除いた集合を表す。

$y \in \mathtt{FV}(N)$ のとき、y は N に自由な出現を持つという。

たとえば、$\mathtt{FV}((\lambda(x:A+B)\mathtt{case}(x,(y:A)\mathtt{inr}(y),(z:B)\mathtt{inl}(u,z)))@(\mathtt{inl}(v))) = \{u,v\}$ であるので、u と v はこの項に自由な出現を持ち、x,y,z 等は、自由な出現を持たない。

定義 11 [変数の対応付け] 変数の対応付けとは、相異なる変数 $x_1,x_2,\cdots,x_n$ をそれぞれ、相異なる変数 $y_1,y_2,\cdots,y_n$ に対応付けるものである。(ここで、x_i を y_i に対応付けている。) この対応付けを $[x_1,x_2,\cdots,x_n/y_1,y_2,\cdots,y_n]$ と書く。なお、x_i 同士、y_i 同士は相異ならなければならないが、$x_i = y_j$ となるものがあってもよい。

定義 12 [対応付けのもとでの α 同値性] 変数の対応付け σ のもとで項 M と項 N が α 同値であることを、$M \simeq_\sigma N$ と書き、次のように定義する。

- $\sigma = [x_1,x_2,\cdots,x_n/y_1,y_2,\cdots,y_n]$ なら、$1 \le i \le n$ となるすべての i に対して、$x_i \simeq_\sigma y_i$
- $\sigma = [x_1,x_2,\cdots,x_n/y_1,y_2,\cdots,y_n]$ で、$x = x_i$ となる i も、$x = y_j$ となる j もないなら、$x \simeq_\sigma x$.
- $L_1 \simeq_\sigma L_2$ かつ $M_1 \simeq_\sigma M_2$ なら、$(L_1@M_1) \simeq_\sigma (L_2@M_2)$
- $\sigma = [x_1,x_2,\cdots,x_n/y_1,y_2,\cdots,y_n]$ かつ、$\tau = [x_1,x_2,\cdots,x_n,u/y_1,y_2,\cdots,y_n,v]$ で、$M_1 \simeq_\tau M_2$ なら、$\lambda(u:A)M_1 \simeq_\sigma \lambda(v:A)M_2$ である。
- $M_1 \simeq_\sigma M_2$ ならば $f(M_1) \simeq_\sigma f(M_2)$ 、ただし、f は 1 引数の関数記号 (`left`, `right`, `inl`, `inr`, `abort`)
- $L_1 \simeq_\sigma L_2$ かつ $M_1 \simeq_\sigma M_2$ なら、$g(L_1,M_1) \simeq_\sigma g(L_2,M_2)$ ただし、g は 2 引数の関数記号 $((\_,\_))$
- $\sigma = [x_1,x_2,\cdots,x_n/y_1,y_2,\cdots,y_n]$ かつ、$\tau = [x_1,x_2,\cdots,x_n,x/y_1,y_2,\cdots,y_n,y]$ かつ $\rho =$

$[x_1, x_2, \cdots, x_n, u/y_1, y_2, \cdots, y_n, v]$ で、$M_1 \simeq_\sigma M_2$ かつ $L_1 \simeq_\tau L_2$ かつ $P_1 \simeq_\rho P_2$ なら、$\mathtt{case}(M_1, (x:A)L_1, (u:B)P_1) \simeq_\sigma \mathtt{case}(M_2, (y:A)L_2, (v:B)P_2)$

定義 13 [α 同値性] 変数の対応付け σ が空のとき (上記の定義で $n = 0$ のとき)、$M \simeq_\sigma N$ であれば M と N は α 同値であると言い、$M \equiv_\alpha N$ と書く。

かなり面倒な定義ではあったが、結果として、α 同値性のこの定義は、我々の直感通りの結果を与える。つまり、項の中の束縛変数を一斉に他の変数 (ただし、他の変数と衝突のないもの) にいれかえたものが α 同値である。

α 同値性を考慮にいれて代入の定義をやりなおそう。以前の $M\{x := N\}$ の定義で、条件にひっかかってしまい代入が定義できなかったケースでは、あらかじめ項 M を α 同値な他の項におきかえることで、条件を満たすようにする。これは、束縛変数の名前を一斉に他のものにおきかえることになる。これにより、代入の条件を満たすようにすることが常に可能となる。

ただし、厳密にやるならば、以下の点をきちんと示す必要があり、これらは読者の演習問題とする。

- α 同値性を保って、束縛変数をいつでも他の変数に変更できる。
- α 同値性で他の項におきかえても、自由変数は変わらない。
- M の部分項を α 同値な別の項におきかえても、α 同値である。
- α 同値な項は一般にたくさんあるが、それのうちのどれを取っても計算結果がかわらない。すなわち、M と M' が α 同値で、N と N' が α 同値ならば、$M\{x := N\}$ と $M'\{x := N'\}$ は α 同値である。

なお、ひとたび α 同値性を導入すると、代入だけでなく、項に対するあらゆる操作・定義において、α 同値な項にいれかえても問題ないことを示す必要がある。これは、実は、コンピュータを用いた定理証明の分野では大問題であり、様々な解が考えられており、「いっそのこと束縛変数をやめてしまえ」という解もある (de Bruijn index, de Bruijn level, SK コンビネータなど)」。

5.7 計算戦略

前章で，型付きラムダ計算の体系に対する計算規則を与えた．しかし，そこでの計算規則は，プログラムの中に，計算可能なパターンがあれば，(それをどこでもいつでも) 計算した結果に置き換えてよい，というものであった．従って，そこでの「計算」の概念は，非決定的 (non-deterministic) であり，1 つのプログラムから多数の計算道筋が発生するものであった．このような道筋の選択方法を「計算戦略」と呼ぶ.

現実のプログラム言語では，代入文，ループや再帰呼び出しなどを含むため，合流性や停止性が成立せず，計算戦略によって答えが異なることがある.

- 代入文 $x = x+1$ と 代入文 $x = x*2$ は，どちらを先に実行するかで結果が異なるので，$(x = x+1, x = x*2)$ は，計算戦略を決めなければプログラムの意味が一意的に定まらない.
- for ループ，while ループ，再帰呼出しがあれば止まらないプログラムを書くことができる．そのようなプログラムを Ω とすると，$(\lambda(x:*)0)@\Omega$ は，引数の計算を先にすると止まらないし，関数への適用 (代入) を先にすると 0 になってすぐ止まる.

そこで，プログラムの意味を定めるためたには、計算戦略をきちんと定式化して扱うことが必要である。

ここでは、代表的な 2 つの計算戦略である値呼び計算と名前呼び計算を取りあげて定式化する。

5.8 値呼び計算

値呼び (call by value, CBV) 計算を，導出の形で定式化する．値呼び計算とは，関数に引数を適用する計算において，引数を先に計算して値 (計算結果) にしてから渡す計算方式である．

この場合の判断は，項 M と値 V に対して，$M \downarrow V$ の形である[*12]．ここで「値」(value) とは，「計算の結果」を表すものであり，項のうちの一部である．ここで扱っている体系では，値は，定数 (整数か true,false)、変数 x, $\lambda(x:A)M$ の形，$\mathtt{inl}(M)$ の形，$\mathtt{inr}(M)$ の形，(M,N) の形に限定される．$M \downarrow V$ の直感的な意味は，「プログラム M を計算すると，その結果は V になる」というものである．

値呼び計算の規則は，型付きラムダ計算の項の種類に応じて以下のように与えられる．

$$\frac{}{x \downarrow x}\ var$$

$$\frac{M \downarrow V \quad N \downarrow W}{(M,N) \downarrow (V,W)}\ pair \qquad \frac{M \downarrow (V,W)}{\mathtt{left}(M) \downarrow V}\ left \qquad \frac{M \downarrow (V,W)}{\mathtt{right}(M) \downarrow W}\ right$$

$$\frac{M \downarrow V}{\mathtt{inl}(M) \downarrow \mathtt{inl}(V)}\ inl \qquad \frac{M \downarrow V}{\mathtt{inr}(M) \downarrow \mathtt{inr}(V)}\ inr$$

$$\frac{M \downarrow \mathtt{inl}(V) \quad N\{x := V\} \downarrow W}{\mathtt{case}(M,(x:A)N,(y:B)L) \downarrow W}\ case1 \qquad \frac{M \downarrow \mathtt{inr}(V) \quad L\{y := V\} \downarrow W}{\mathtt{case}(M,(x:A)N,(y:B)L) \downarrow W}\ case2$$

$$\frac{}{\lambda(x:A)M \downarrow \lambda(x:A)M}\ lambda \qquad \frac{M \downarrow \lambda(x:A)L \quad N \downarrow V \quad L\{x := V\} \downarrow W}{M@N \downarrow W}\ apply$$

この規則で注目すべきは，apply 規則において値呼びを実現していること (引数 N をそのまま代入しているのではなく，N を計算して値 V にしてから代入している) である．

また，$M\{x := W\}$ という表現は，「項 M の中の変数 x に項 W を代入したもの」を表す．代入の定義は，「型のないラムダ計算」における代入の定義とほぼ同様であるので (型の情報がはいっているかどうかの違いだけ) ここでは省略する．なお，この代入のことを，$M[W/x]$ と書くこともある．これを $M[x/W]$ とは書かないことに注意せよ．

5.9 名前呼び計算

値呼び計算が定式化された後では，名前呼び (call by name, CBN) 計算を定式化するのは容易である．判断は，先ほどと同様，$M \downarrow V$ の形であり，規則はほとんど値呼びと同様であるが，apply 規則だけが異なる．

$$\frac{M \downarrow \lambda(x:A)L \quad L\{x := N\} \downarrow W}{M@N \downarrow W}\ apply$$

引数 N を計算せず，そのまま代入しているので名前呼びになっている．

[*12] 計算の対象となる M は，当然ながら，ある Γ, A に対して $\Gamma \vdash M : A$ が導出できなければいけない．したがって，$M \downarrow V$ は本来なら $\Gamma \vdash M \downarrow V : A$ と書いた方が良いものである．ここでは，書く手間を減らして簡便な表記をとった．

5.10　型付きラムダ計算の体系の性質

型付きラムダ計算のような計算体系に対して，成立が期待される主な性質には以下の 3 つがある[*13]．

- 型システムの健全性…型が整合的なプログラムを計算するとき，計算の途中の任意の時点 (計算終了後を含む) で，型が整合的であるという性質．
- 合流性…どのような順番で計算を行ったとしても，計算結果が一意的であるという性質．
- 停止性 (強い停止性，弱い停止性)…
  どのような順番で計算を行ったとしても，計算がいつか必ず停止するという性質 (強い停止性)，ある (適当な) 順番で計算を行ったときに，計算がいつか必ず停止するという性質 (弱い停止性)．

これらの性質が成立することにより，どのような「良い」ことがいえるかは後の章で考えることにして，さしあたり，これらの性質を数学的に厳密に述べ，証明を考えることにする．なお，証明の詳細について完全に理解することは本講義の範囲をこえるので，以下で述べる定理の主張のみを理解する程度でよい (どういう定理かは理解すべきであるが，証明は理解できなくてもよい．)

5.11　型システムの健全性

定理 7（サブジェクトリダクション (Subject Reduction))　$\Gamma \vdash M : A$ が (前に述べた推論規則により) 導出可能であり，$M \to^* N$ であれば，$\Gamma \vdash N : A$ が導出可能である．

ここで $M \to^* N$ は，$M \to N$ の反射的・推移的閉包であり，0 ステップ以上の計算により M が N に変形できることを意味している．また，以下の証明で $M \to_k^* N$ という記法を使うが，これは k ステップの計算で M が N に変形できることを意味している．

証明の概要: まず，以下の性質を $\phi(k)$ とする．これは，ちょうど k ステップで M から N に簡約 (計算) 可能である場合に限定した定理の内容である．

$\Gamma \vdash M : A$ が導出可能であり，$M \to_k^* N$ であれば，$\Gamma \vdash N : A$ が導出可能である．

「任意の自然数 k に対して $\phi(k)$ が成立する」ことを証明できれば定理は証明されたことになる．これを，k に関する数学的帰納法で証明する．すなわち，以下の 2 つを証明できればよい．

- $\phi(0)$ を証明する．
- $\phi(k)$ が成立することを仮定して $\phi(k+1)$ を証明する．

$\phi(0)$ は，この場合，trivial に成立する内容である．「$\phi(k)$ ならば $\phi(k+1)$」 を証明するためには，以下の性質 (これを性質 (*) と呼ぶことにする) が証明できればよい．

$\Gamma \vdash M : A$ が導出可能であり，$M \to N$ であれば，$\Gamma \vdash N : A$ が導出可能である．

これはちょうど 1 ステップの計算で型が保たれる，という性質である．この性質を証明するために，もう一度帰納法を使う．今度は自然数に関する帰納法ではなく，「$\Gamma \vdash M : A$ の導出」の構成に関する帰納法 (木に

[*13] ここでの説明は，わかりやすさのために数学的厳密さを若干犠牲にしている．

関する帰納法) を使う．この帰納法を詳しく書くと，以下のようになる．

- $\Gamma \vdash M : A$ の最終導出規則が var 規則であり，そのすべての部分導出に対して性質 (*) が成立し，さらに，$M \to N$ であれば，$\Gamma \vdash N : A$ が導出可能である．
- 上記の var を λ で置きかえたもの
- 上記の var を apply で置きかえたもの
- 上記の var を pair で置きかえたもの
- 上記の var を left で置きかえたもの
- 上記の var を right で置きかえたもの
- 上記の var を inl で置きかえたもの
- 上記の var を inr で置きかえたもの
- 上記の var を case で置きかえたもの

「これら 9 つが全て証明できれば，任意の導出 $\Gamma \vdash M : A$ に対して性質 (*) が成立する」という推論法が，「導出の構成に関する帰納法」である[*14]．

上記 9 つの項目を見てみると，たとえば，*var* 規則の場合は，「部分導出」はないので，以下のことを証明すればよい．

> M が変数 x であり，$x : A$ が Γ に含まれていて，さらに，$M \to N$ であれば，$\Gamma \vdash N : A$ が導出可能である．

この場合 $x \to N$ となる N は存在しないので，仮定が成立せず，したがって，全体としては trivial に成立する．

また，*pair* 規則の場合は，「部分導出」が 2 つあるので，以下のことを証明すればよい．

> M が項 (L, N) であり，A が型 $B \times C$ であり，$\Gamma \vdash L : B$ の導出が存在して，かつ，その導出に対して性質 (*) が成立し，$\Gamma \vdash N : C$ の導出が存在して，かつ，その導出に対して性質 (*) が成立し，さらに，$M \to N$ であれば，$\Gamma \vdash N : A$ が導出可能である．

これは，以下のようにして簡単に証明できる．$(L, P) \to N$ ということから，N が (L_1, P) の形で，かつ，$L \to L_1$ であるか，または，N が (L, P_1) の形で，かつ，$P \to P_1$ であるかのいずれかである．前者の場合，$\Gamma \vdash L : B$ の導出に対して性質 (*) が成立するので，$\Gamma \vdash L_1 : B$ という導出を作ることができる．そこで，この導出と，もともとあった $\Gamma \vdash P : C$ の導出を pair 規則により組み合わせれば，$\Gamma \vdash (L_1, P) : B \times C$ の導出を作ることができる．後者の場合も同様である．

ほかの規則の場合も同様に証明できる．(ただし，apply 規則や case 規則の場合に「導出に対する代入」という概念を必要とするので，かなり面倒な議論が必要になる．その詳細は省略する．) [証明の概要おわり]

定理 8（canonical form に関する性質） $\vdash M : A$ が導出可能であれば，M は計算可能であるか，または，canonical form (最終規則がλ , pair, inl, inr のいずれかで導出された項) である．

$\vdash M : A$ が導出可能である，ということは，M が型がつく項であるだけでなく，$\Gamma = \{\}$ ということを意味している．つまり，M は変数の自由出現を持たない項である．通常「プログラム」は，自由な変数出現を持

[*14] これは，n 分木に対する帰納法の一般化である．

たない項であるが，この定理は，「プログラム」の計算結果は，canonical form であることを意味している．

逆にいうと，計算結果 (canonical form) 以外の形で計算が進まなくなることはない (計算途中でひっかかってしまう) ことがないことを意味している．

5.12 合流性

定理 9（合流性） 型つきラムダ計算の項 M, N_1, N_2 に対して (つまり，M, N_1, N_2 は適当な宣言 $\Gamma_1, \Gamma_2, \Gamma_3$ と型 A_1, A_2, A_3 に対して，$\Gamma_1 \vdash M : A_1, \Gamma_2 \vdash N_1 : A_2, \Gamma_3 \vdash N_2 : A_3$ が導出可能であるとする)，$M \to^* N_1$ かつ $M \to^* N_2$ であれば，ある項 L があって，$N_1 \to^* L$ かつ $N_2 \to^* L$ となる．

この定理は，Church-Rosser 定理とも呼ばれる[*15]．型のないラムダ計算に対しても成立する重要な定理であり，計算の順序によらず結果が一意的であることを意味している．

この定理の証明は省略する．

5.13 停止性

定理 10（強い停止性） $\Gamma \vdash M : A$ が推論できるなら，$M \to N_1 \to N_2 \to N_3 \to \cdots$ となる無限の計算列は存在しない．

これは，型がきちんとつく項 M から始めた計算は，どのように計算しても，必ず有限回の計算により結果 (値) に到達することを意味している．

定理 11（弱い停止性） $\Gamma \vdash M : A$ が推論できるなら，これ以上計算可能でない N に対して，$M \to^* N$ となる．

これは，型がきちんとつく項 M から始めた計算は，うまくやれば，有限回の計算により結果 (値) に到達することを意味している．

明らかに，強い停止性が成立すれば，弱い停止性は成立する．本講義でのべた型付きラムダ計算の体系に対しては，強い停止性が成立するので弱い停止性も成立する．一方，型のないラムダ計算の体系では，(強い/弱い) 停止性は成立しない．実際，$(\lambda x.xx)@(\lambda x.xx)$ という項の計算は無限ループとなる．この定理の証明は省略する．

現在利用可能なほとんど全てのプログラム言語に対して，「停止しないプログラム」を書くことが可能であるという事実からすると，型付きラムダ計算の体系が停止性を満たす，という定理は奇異にうつるかもしれない．しかし，通常，人間は，停止しないことを目的にプログラムを書くことはないだろう．「どんな入力に対しても有限時間内に停止して答えを出す」プログラムを書きたいのがほとんどの場合である[*16]．そのような観点からすると，「プログラム言語の機能のうち，この部分はきちんと停止する機能である」「停止しないのは，この機能である」という風にきちんと区別できることが好ましい．区別することにより，「このプログラムが停止するためには，どの部分をチェックすればよいか」が明確になるからである．

[*15] Church も Rosser も人の名前であり，この分野を築いた偉人たちである．

[*16] OS のカーネルやネットワーク・サーバなど，無限に動き続けるのが普通であるようなプログラムもある．しかし，これらのプログラムでも，リクエストに対しては有限時間内に応答することが求められているのであり，そのような応答をモデル化する場合は，やはり，停止性が満たされる必要がある．

言いかえると，多くのプログラミング言語は，型付きラムダ計算を基礎として，それに機能を追加する形で理解することができる．たとえば，「代入によって値を変更できる変数」，「再帰呼び出し」「繰り返し (C 言語の for 文など)」「ファイルへの入出力」などを加えると，現実的なプログラミング言語にかなり近付けることができる．そのとき，型の健全性，停止性，合流性などの性質がどのようになるか (保たれるのか，破られるのか) を観察することにより，プログラミング言語の構造が良く見えてくるのである．

6 関数型プログラム言語の体系

前節の型付きラムダ計算の体系は、命題論理の体系と対応が良いという利点があり、型付けや計算など、計算体系を考える基礎としての役割を担っていた。一方で、基礎的すぎるという欠点があり、OCaml, SML, Haskell など現実的なプログラミング言語と比べると機能が不足しており、たとえば、整数や整数上の演算 (足し算など) がない、関数の再帰的定義ができない、など、有用なプログラムを書くことができなかった。

本節では、単純型付きラムダ計算を、より現実的なプログラミング言語に近付けた計算体系 CoreML を導入する。計算体系 CoreML は、整数型、真理値型やその上の演算、再帰関数の定義などの機能を持っている。また、前節では Church 流の体系 (束縛変数の型を明示する方式) を学習したので、ここでは Curry 流の体系 (束縛変数の型を明示しない) を採用する。結果として、CoreML は、ML 系言語 (OCaml, SML) のコア部分に相当する体系である。

6.1 構文と型付け

型の構文は、単純型付きラムダ計算の型の構文とほぼ同じだが、基本となる型は、抽象的な K_i でなく、具体的な、`int`, `bool` の 2 つとする。また、簡単のため、直和型は削除した。よって型の構文は以下の規則で定められる。

$$\frac{}{\texttt{int} : Type} \qquad \frac{}{\texttt{bool} : Type} \qquad \frac{A : Type \quad B : Type}{A \times B : Type} \qquad \frac{A : Type \quad B : Type}{A \to B : Type}$$

次に、項の構文である。上述したように、ここでは Curry 流の構文を導入する。すなわち、$\lambda(x : A)M$ のように x の型 A を明示するのではなく、$\lambda x.M$ のように、x の型を明示しない流儀である。プログラム言語 OCaml では、これは `fun x -> M` と記述される。

以下に、項とその型付けを定める規則を与える。

$$\frac{((x : A) \in \Gamma)}{\Gamma \vdash x : A}\ \texttt{var} \qquad \frac{(n \text{ は整数定数})}{\Gamma \vdash n : \texttt{int}}\ \text{int} \qquad \frac{(b \text{ は真理値定数})}{\Gamma \vdash b : \texttt{bool}}\ \text{bool}$$

$$\frac{\Gamma \vdash M : \texttt{int} \quad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M + N : \texttt{int}}\ \text{plus} \qquad \frac{\Gamma \vdash M : \texttt{int} \quad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M = N : \texttt{bool}}\ \text{eq} \qquad \frac{\Gamma \vdash M : \texttt{int} \quad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M > N : \texttt{bool}}\ \text{comp}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B}\ \texttt{pair} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \texttt{left}(M) : A}\ \texttt{left} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \texttt{right}(M) : B}\ \texttt{right}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \text{lambda} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ @\ N : B}\ \text{apply}$$

$$\frac{\Gamma \vdash L : \texttt{bool} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \texttt{if}\ L\ \texttt{then}\ M\ \texttt{else}\ N : A}\ \text{if}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : B}\ \text{let} \qquad \frac{\Gamma, f : A \to B, x : A \vdash M : B}{\Gamma \vdash \texttt{fix}\ f.x.M : A \to B}\ \text{fix}$$

ここで、int 規則と bool 規則はそれぞれの型の定数を導入する規則である。たとえば、3 や -5 は整数定数、`true`と `false`が真理値定数である。これらに対する演算として、加算 (plus)、等号 (eq)、大小比較 (comp) をいれる。減算や乗算などをこの体系に入れる場合も、同様の型付け規則となる。

直積型に関する規則 pair, left, right は前節と同じである。また、関数型に関する規則 lambda, apply も前節とほぼ同じであるが、lambda において、束縛される変数 x に型を記載しない点が異なる。

if 規則は、多くのプログラム言語の if-then-else と同様である。

let は多くの関数型プログラム言語が持つ構文であり、let x=M in N において、まず M を計算し、その結果を変数 x に束縛して、N を計算するというものである。つまり、局所的な束縛を導入する。この節の言語 (多相型を持たない言語) の範囲では、let x=M in N は $(\lambda x.N)M$ と、型付けの点でも計算結果の点でも全く等価である。後に多相型に拡張するとき、let は独自の意味を持つ。

fix は再帰的関数を定義するためのものであり、fixpoint operator (不動点演算子) と呼ばれるものである。$\texttt{fix}\ f.x.M$ の意味は、$f(x) = M$ という定義で導入される関数 f ということである。ここで、M には (x だけでなく)f も現れてよく、現れた場合は、再帰関数となる。(f が現れない場合は、再帰関数ではないので、$\lambda x.M$ と同じである。)

OCaml 言語では、let rec f x = M in N という構文で、再帰関数 f を定義することができるが、$\texttt{fix}\ f.x.M$ はこれに対応している。

細かい点についての補足: OCaml の let rec と違って、$\texttt{fix}\ f.x.M$ は、f という関数の定義ではなく、再帰関数そのものを表す。たとえば、$\texttt{fix}\ f.x.f@x+1$ と $\texttt{fix}\ g.x.g@x+1$ は α 同値であり、同じ再帰関数を表す。そして、これらを後で「関数 f」とか「関数 g」として参照することはできない。よって、OCaml の let rec f x = M in N と等価な式は、このテキストの体系では、$\texttt{let}\ f = (\texttt{fix}\ f.x.M)\ \texttt{in}\ N$ となる。1 つ目の f のスコープは N だけであり、2 つ目の f のスコープは M だけである。

例 14（型付けの例 1） $M = \texttt{fix}\ f.x.\texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else}\ f@(x + (-1)) + x$ の型付けは以下の通り．(ここで $\Gamma = f : \texttt{int} \to \texttt{int},\ x : \texttt{int}$ と置いた．)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \texttt{int}} \quad \dfrac{}{\Gamma \vdash 0 : \texttt{int}}}{\Gamma \vdash x = 0 : \texttt{bool}} \quad \dfrac{}{\Gamma \vdash 0 : \texttt{int}} \quad \dfrac{\dfrac{}{\Gamma \vdash f : \texttt{int} \to \texttt{int}} \quad \dfrac{\dfrac{}{\Gamma \vdash x : \texttt{int}} \quad \dfrac{}{\Gamma \vdash -1 : \texttt{int}}}{\Gamma \vdash x + (-1) : \texttt{int}}}{\Gamma \vdash f@(x + (-1)) : \texttt{int}} \quad \dfrac{}{\Gamma \vdash x : \texttt{int}}}{\Gamma \vdash f@(x + (-1)) + x : \texttt{int}}}{\Gamma \vdash \texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else}\ f@(x + (-1)) + x}}{\vdash M : \texttt{int} \to \texttt{int}}$$

例 15（型付けの例 2） 以下の項 M の型付け (の一部) は以下の通り。

$$\texttt{fix}\ f.x.\texttt{if}\ \texttt{left}(x) = 0\ \texttt{then}\ \texttt{true}\ \texttt{else if}\ \texttt{right}(x) = 0\ \texttt{then}\ \texttt{false}\ \texttt{else}\ f@(\texttt{left}(x) - 1, \texttt{right}(x) - 1)$$

(ここで $\Gamma = f : \texttt{int} \times \texttt{int} \to \texttt{bool},\ x : \texttt{int} \times \texttt{int}$ と置いた．)

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \texttt{left}(x) = 0 : \texttt{bool}} \quad \dfrac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \quad \dfrac{\dfrac{}{\Gamma \vdash f : \texttt{int} \times \texttt{int} \to \texttt{bool}} \quad \dfrac{\vdots}{\Gamma \vdash (\texttt{left}(x) - 1, \texttt{right}(x) - 1) : \texttt{int} \times \texttt{int}}}{\Gamma \vdash f@(\texttt{left}(x) - 1, \texttt{right}(x) - 1) : \texttt{bool}}}{\Gamma \vdash \texttt{if}\ \texttt{left}(x) = 0\ \texttt{then}\ \texttt{true}\ \texttt{else if}\ \texttt{right}(x) = 0\ \texttt{then}\ \texttt{false}\ \texttt{else}\ f@(\texttt{left}(x) - 1, \texttt{right}(x) - 1) : \texttt{bool}}}{\vdash M : \texttt{int} \times \texttt{int} \to \texttt{bool}}$$

次の節で計算を定式化する前に、「項 M が閉じていて型がつく」という言葉を定義する。これは、ある型 A に対して、上述の型付け規則により $\vdash M : A$ が導けることである。ここで $\vdash$ の左に変数宣言が 1 つもないが、これは M に自由変数がない (M が閉じている) ことを表す。コンピュータ上の言語処理系 (コンパイラ等) で処理するのは、通常、閉じていて型がつく項だけであり (そうでなければ、unfound variable や type error といったコンパイル時のエラーとなる)、その意味で、「閉じていて型がつく項」を単に「プログラム」と呼んで、一般の項と区別することがある。

6.2 計算規則: 値呼び計算の定式化

この言語に対して、値呼び計算を定式化する。

前節と同様の定式化をすることも可能ではあるが、ここでは、代入は使わず、そのかわりに環境をつかった形で定式化する。

まず、以下の形の項 V を、値 (あたい、value) と呼び、以下の形の表現 E を環境[*17]と呼ぶ。

$$V ::= n \mid b \mid (V, V) \mid \mathtt{clos}(x, M, E) \mid \mathtt{rclos}(f, x, M, E)$$
$$E ::= \mathtt{[]} \mid E\mathtt{[}x \mapsto V\mathtt{]}$$

ここで n は整数定数、b は真理値定数である。

上記の定義で、$\mathtt{clos}(x, M, E)$ は「関数クロージャ (関数閉包)」と呼ばれるものであり、ラムダ式と環境を 1 セットにまとめたものである。これは、言語 CoreML の構文の定義になかったものであり、「プログラムとしてユーザは記述できないが、プログラムの実行中に出現するもの (実行時の値)」の一種である。関数クロージャは、静的束縛をする関数型言語で必要なものであり、その詳細は、この授業の講義の説明を参照されたい。(春学期の「プログラム言語論」の講義資料でも詳述している。)

$\mathtt{rclos}(f, x, M, E)$ は、再帰関数に関する関数クロージャ (先頭の r は、「再帰」を表す) である。すなわち、関数閉包 $\mathtt{clos}(x, M, E)$ は、再帰のない関数、つまり、$\lambda x.M$ という項を評価したときに生成されるが、$\mathtt{rclos}(f, x, M, E)$ は、fix を用いた再帰関数を評価したときに生成される。

環境 E は、変数 x を値 V に対応付ける写像である。$\mathtt{[]}$ は空の対応をあらわす、$E\mathtt{[}x \mapsto V\mathtt{]}$ は、環境 E に「x を V に対応付ける」機能を追加してできる新しい環境をあらわす。

環境 E のもとでの変数 x の値 $\mathtt{lookup}(x, E)$ を以下のように定義する。

$$\mathtt{lookup}(x, \mathtt{[]}) \stackrel{\mathrm{def}}{=} (\text{undefined})$$
$$\mathtt{lookup}(x, E\mathtt{[}x \mapsto V\mathtt{]}) \stackrel{\mathrm{def}}{=} V$$
$$\mathtt{lookup}(x, E\mathtt{[}y \mapsto V\mathtt{]}) \stackrel{\mathrm{def}}{=} \mathtt{lookup}(x, E) \quad \text{if } x \neq y$$

この定義からわかるように、E で、同じ変数 x が 2 回以上束縛された場合は、後から追加された方 (E の表記では後ろの方) が優先される。また、E の中で x が束縛されていなければ、$\mathtt{lookup}(x, E)$ は未定義 (undefined) である。

細かい補足: 上記では、型がつくかどうかに関係なく「値」を定義した。たとえば、$\mathtt{clos}(x, x@x, \mathtt{[]})$ は、$x@x$ の部分に型が付かないため、(閉じて型がつく項から計算を始めれば) 計算の途中で決して現れることのない値である。よって、このような「おかしな項」は、排除した方が理論的にはすっきりとする。このためには、関数クロージャと再帰関数クロージャに対する型付け規則が必要であるが、そ

[*17] 型付け時の環境 Γ と区別して、この E を実行時環境と呼ぶことがある。

れらを型付けするためには、環境の型付けも必要になる。これは若干ややこしいのと、計算規則を理解するためには、型付けを無視して読んでも差し支えないので、あとまわしにする。

次に、体系 CoreML における計算を表す評価関係 $\downarrow$ を次の規則で与える。ただし、基本的判断は

$$E \rhd M \downarrow V$$

の形で、かつ、M が型をもつ項であり、V が値の場合に限定する。(M が型をもたない場合は、プログラムとは見なせず、一種のコンパイラエラーとなるので、それを評価することはない。ただし、計算の途中で M が自由変数を含むことはあり得るので、M を「閉じた項」に限定することはしない。)

$E \rhd M \downarrow V$ は、実行時環境 E のもとでプログラム M を評価 (計算) すると、値 V を得る、と読む。

この言い回しからわかるように、これは 1 ステップの計算ではなく、「計算できる限り計算を続行する」という計算をあらわす。つまり、M を計算すると無限ループになるとき、どんな V に対しても、$E \rhd M \downarrow V$ とはならない。

$$\frac{(\texttt{lookup}(x, E) = V \text{ の時})}{E \rhd x \downarrow V}\ \text{var} \qquad \frac{(n \text{ が整数定数のとき})}{E \rhd n \downarrow n}\ \text{int} \qquad \frac{(b \text{ が真理値定数のとき})}{E \rhd b \downarrow b}\ \text{bool}$$

$$\frac{E \rhd M \downarrow V_1 \quad E \rhd N \downarrow V_2 \quad (V_1 + V_2 = V \text{ となるとき})}{E \rhd M + N \downarrow V}\ \text{plus}$$

$$\frac{E \rhd M \downarrow V_1 \quad E \rhd N \downarrow V_2 \quad (V_1 > V_2 \text{が成立する時})}{E \rhd M > N \downarrow \texttt{true}}\ \text{comp1}$$

$$\frac{E \rhd M \downarrow V_1 \quad E \rhd N \downarrow V_2 \quad (V_1 > V_2 \text{が不成立の時})}{E \rhd M > N \downarrow \texttt{false}}\ \text{comp2}$$

$$\frac{E \rhd M \downarrow V_1 \quad E \rhd N \downarrow V_2 \quad (V_1 = V_2 \text{が成立する時})}{E \rhd M = N \downarrow \texttt{true}}\ \text{eq1}$$

$$\frac{E \rhd M \downarrow V_1 \quad E \rhd N \downarrow V_2 \quad (V_1 = V_2 \text{が不成立の時})}{E \rhd M = N \downarrow \texttt{false}}\ \text{eq2}$$

$$\frac{E \rhd M \downarrow \texttt{true} \quad E \rhd N \downarrow V}{E \rhd \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ L \downarrow V}\ \text{if1} \qquad \frac{E \rhd M \downarrow \texttt{false} \quad E \rhd L \downarrow V}{E \rhd \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ L \downarrow V}\ \text{if2}$$

$$\frac{E \rhd M \downarrow V \quad E \rhd N \downarrow W}{E \rhd (M, N) \downarrow (V, W)}\ \text{pair} \qquad \frac{E \rhd M \downarrow (V, W)}{E \rhd \texttt{left}(M) \downarrow V}\ \text{left} \qquad \frac{E \rhd M \downarrow (V, W)}{E \rhd \texttt{right}(M) \downarrow W}\ \text{right}$$

$$\frac{}{E \rhd \lambda x.M \downarrow \texttt{clos}(x, M, E)}\ \text{lambda} \qquad \frac{}{E \rhd \texttt{fix}\ f.x.M \downarrow \texttt{rclos}(f, x, M, E)}\ \text{fix}$$

$$\frac{E \rhd M \downarrow V' \quad E[x \mapsto V'] \rhd N \downarrow V}{E \rhd \texttt{let}\ x = M\ \texttt{in}\ N \downarrow V}\ \text{let}$$

$$\frac{E \rhd M \downarrow \mathtt{clos}(x, L, E') \quad E \rhd N \downarrow V \quad E'[x \mapsto V] \rhd L \downarrow V'}{E \rhd M@N \downarrow V'} \text{ apply1}$$

$$\frac{E \rhd M \downarrow \mathtt{rclos}(f, x, L, E') \quad E \rhd N \downarrow V \quad E'[x \mapsto V][f \mapsto \mathtt{rclos}(f, x, L, E')] \rhd L \downarrow V'}{E \rhd M@N \downarrow V'} \text{ apply2}$$

`fix` $f.x.M$ という項は、$f(x) = M$ という再帰的関数定義に対応し、引数に適用すると、再帰的に f のところに `fix` $f.x.M$ 自身を代入する点が、通常の $\lambda x.M$ の形の式との違いである。

上記の規則は一般の E に対して定めたが、計算を始めるときは、閉じた項 M に対する計算としたい。つまり、閉じて型がつく項 M に対して、$[] \rhd M \downarrow V$ が上記の規則で推論できるとき、$eval(M) = V$ と定義し、「プログラム M の計算結果が V である」と言う。

なお、上記の定義をよく見るとわかるように、適用可能な規則はたかだかひとつであるので、この計算は決定的 (deterministic) である。

例 16 非負の整数同士の足し算の関数は、$\lambda y.\mathtt{fix}\ f.x.\mathtt{if}\ x = 0\ \mathtt{then}\ y\ \mathtt{else}\ (f@(x-1)) + 1$ と定義できる。

この項を M とすると、たとえば、$[] \rhd (M@3)@0 \downarrow 3$ が、上記の規則で推論できる。

$$\dfrac{\dfrac{\overline{[] \rhd M \downarrow c_1} \quad \overline{[] \rhd 3 \downarrow 3} \quad \overline{[][y \mapsto 3] \rhd m_1 \downarrow r_1}}{[] \rhd M@3 \downarrow r_1} \quad \overline{[] \rhd 0 \downarrow 0} \quad \dfrac{\dfrac{\overline{E_1 \rhd x \downarrow 0} \quad \overline{E_1 \rhd 0 \downarrow 0}}{E_1 \rhd x = 0 \downarrow \mathtt{true}} \quad \overline{E_1 \rhd y \downarrow 3}}{E_1 \rhd \mathtt{if}\ x = 0\ \mathtt{then}\ y\ \mathtt{else}\ f@(x-1) + 1 \downarrow 3}}{[] \rhd (M@3)@0 \downarrow 3}$$

ここで m_1 等は以下のようにおいた。

$$\begin{aligned} m_1 &= \mathtt{fix}\ f.x.\mathtt{if}\ x = 0\ \mathtt{then}\ y\ \mathtt{else}\ (f@(x-1)) + 1 \\ m_2 &= \mathtt{if}\ x = 0\ \mathtt{then}\ y\ \mathtt{else}\ (f@(x-1)) + 1 \\ c_1 &= \mathtt{clos}(y, m_1, []) \\ r_1 &= \mathtt{rclos}(f, x, \mathtt{if}\ x = 0\ \mathtt{then}\ y\ \mathtt{else}\ (f@(x-1)) + 1, [][y \mapsto 3]) \\ E_1 &= [][y \mapsto 3][x \mapsto 0][f \mapsto r_1] \end{aligned}$$

と置いた。

同様に、$[] \rhd (M@3)@1 \downarrow 4$ の推論の概形は以下の通りである。

$$\dfrac{\begin{array}{c}\vdots \\ [] \rhd M@3 \downarrow r_1\end{array} \quad \overline{[] \rhd 1 \downarrow 1} \quad \dfrac{\begin{array}{c}\vdots \\ E_2 \rhd x = 0 \downarrow \mathtt{false}\end{array} \quad \dfrac{\dfrac{\overline{E_2 \rhd f \downarrow r_1} \quad \begin{array}{c}\vdots \\ E_2 \rhd x - 1 \downarrow 0\end{array} \quad \begin{array}{c}\vdots \\ E_3 \rhd m_2 \downarrow 3\end{array}}{E_2 \rhd f@(x-1) \downarrow 3} \quad \overline{E_2 \rhd 1 \downarrow 1}}{E_2 \rhd f@(x-1) + 1 \downarrow 4}}{E_2 \rhd \mathtt{if}\ x = 0\ \mathtt{then}\ y\ \mathtt{else}\ f@(x-1) + 1 \downarrow 4}}{[] \rhd (M@3)@1 \downarrow 4}$$

ただし、

$$E_2 = [][y \mapsto 3][x \mapsto 1][f \mapsto r_1]$$
$$E_3 = [][y \mapsto 3][x \mapsto 1][f \mapsto r_1][x \mapsto 0][f \mapsto r_1]$$

である。環境 E_3 を見ると、x や f への束縛が 2 回あるが、これは再帰呼び出しを 2 回行ったことに対応している、環境 E_3 での x の値は、後で束縛した 0 の方が取られる (後から束縛したものが優先である。)

問題 1。上記の、$[] \rhd (M@3)@1 \downarrow 4$ の推論図を完成させよ。

問題 2。上と同様に、$[] \rhd (M@3)@2 \downarrow 5$ の推論を行いなさい。

問題 3。M の型付けを行いなさい。

例 17 以下の項は、非負整数 a, b に対して (a, b) の形の引数を 1 つ受け取ると、$a \leq b$ ならば `true` を、$a > b$ ならば `false` を返す関数である。ただし、$a, b \geq 0$ とする。(a, b に負のものがあったら、答えを返さないことがある。)

$\texttt{fix}\, f.x.\texttt{if}\ \texttt{left}(x) = 0\ \texttt{then true else if}\ \texttt{right}(x) = 0\ \texttt{then false else}\ f(\texttt{left}(x) - 1, \texttt{right}(x) - 1)$

上記の項を N とすると、$N@(2,1)$ は `false`であり、$N@(1,2)$ は `true`となる。$N@(-1,-2)$ は定義されない。(計算が止まらない。)

問題 1。$[] \rhd N@(2,1) \downarrow \texttt{false}$ および $[] \rhd N@(1,2) \downarrow \texttt{true}$ を推論せよ。

問題 2。N の型付けを行いなさい。

演習問題: 以下の関数を、体系 CoreML のプログラム (項) として表現し、ユニットテストを (紙の上で) 行いなさい。ただし、この場合のユニットテストとは、関数に対して具体的な入力例をあたえ、期待した出力が得られるかどうかをテストするものである。また、関数の厳密な仕様は各自が適当に決めてよい。(たとえば、入力となる引数が 2 つある場合、(a, b) という 1 つの引数にするか、a をもらってから b をもらう、という Curry 化関数の考えで実装するかは各自が決めてよい。また、非負の整数上で動けばよいが、負の整数のことも考慮した場合はボーナス点を与える。)

- 引き算をする関数 sub
- かけ算をする関数 mult
- 最大公約数を計算する関数 gcd

可能ならば、型付けや、計算に関する推論も行なうとよいが、紙と鉛筆でそれをやるととても大変であるので、計算機上のシステムを使った演習課題として残しておくことにする。(実は、計算機上でやっても、計算に関する推論は、ステップ数が非常に多くなって大変である。)

7　型推論

第 5 章で見たように、プログラムの型に関して、微妙に異なるいくつかの判定問題がある。その主要なものを再掲すると以下の通りである。

- Γ, M, A が与えられたとき，$\Gamma \vdash M : A$ が導けるか？ (type checking, 型検査問題)
- M が与えられたとき，$\Gamma \vdash M : A$ が導ける Γ, A があるか？ (type inference, 型推論問題その 1)
- Γ, M が与えられたとき，$\Gamma \vdash M : A$ が導ける A があるか？ (type inference, 型推論問題その 2)

本講義で扱う体系のうち、単純型付きラムダ計算と関数プログラミングの体系では、上記の判定問題が決定可能となり、それぞれに対して，判定を行なう (有限時間で必ず停止する) アルゴリズムが存在する。また、型推論問題では、単に「型 A 等があるかどうか」ではなく、「ある」場合には、具体的な A 等を返すアルゴリズムも存在する。

これらすべてを紹介すると膨大になるので、ここでは、関数プログラミングの体系に対する型推論のアルゴリズムについて学習する。

7.1　型変数の導入

計算体系 CoreML の型で特徴的なことは、1 つの項が複数の型を持つことである。

たとえば、$\lambda f.\lambda x.\ f@(f@x)$ という項は、$(\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})$ という型を持つが、$(\mathtt{bool} \to \mathtt{bool}) \to (\mathtt{bool} \to \mathtt{bool})$ という型も持つ。一般に、$(\square \to \square) \to (\square \to \square)$ という形の任意の型を持つ。

また, $\lambda x.\,(\mathtt{right}(x), \mathtt{left}(x))$ という項は、$(\mathtt{int} \times \mathtt{bool}) \to (\mathtt{bool} \times \mathtt{int})$ という型を持つが、$(\mathtt{int} \times \mathtt{int}) \to (\mathtt{int} \times \mathtt{int})$ という型も持つ。もっと一般に、$(\square_1 \times \square_2) \to (\square_2 \times \square_1)$ という形の任意の型を持つ。

どちらのケースでも，「一般に」と書いた型は，型のパターンであり，それらの $\square$ 等の中に任意の型をいれることにより，具体的な型がでてくる．

型推論アルゴリズムにおいては，$\lambda f.\lambda x.\ f(f(x))$ という項に対して，$(\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})$ のような特定の型ではなく，$(\square \to \square) \to (\square \to \square)$ という型のパターンを求めたい．このような，型のパターンにおける穴 $(\square)$ をあらわすため、型変数 $\alpha, \beta, \gamma, \cdots$ を用いる．

よって，本節においては、型は、 CoreML の定義に型変数を加えたものとする。上記の 2 つの型パターンは、型変数を使った型として、$(\alpha \to \alpha) \to (\alpha \to \alpha)$ および $(\alpha \times \beta) \to (\beta \times \alpha)$ と表すことができる。なお、型変数をどう選んでも，型のパターンとしては同じなので，1 つ目のものを、$(\beta \to \beta) \to (\beta \to \beta)$ などと表してもよい．

このような「型変数を含むかもしれない型」、つまり、型のパターンのことを、正式には、型スキーム (type scheme) と呼ぶ。ただし、本章では、これがたくさんでてくるので、型変数を含むかもしれない型を、単に「型」と呼ぶことにしよう。

本章における型の定義:

$$A, B ::= \alpha \mid \mathtt{int} \mid \mathtt{bool} \mid A \times B \mid A \to B$$

ただし，α は型変数をあらわし，型変数は無限個あるとする．

7.2 型変数に対する代入

型変数は、上記の型パターンにおける □ のように、「そこに、具体的な型を埋めるための案」をあらわす。この「埋める」操作は、型変数に対する代入と呼ばれ、以下のように定義できる。

定義 14 [型変数に対する代入] 型変数に対する代入 (単に「型代入」とも言う) は，いくつかの型変数 $\alpha_1, \alpha_2, \cdots, \alpha_n$ (ただし、α_i は互いに相異なる)、および、それと同じ個数の (型変数を含むかもしれない) 型 $A_1, A_2, \cdots, A_n$ に対して，$[\alpha_1 := A_1, \alpha_2 := A_2, \cdots \alpha_n := A_n]$ の形をした表現である．

ただし、$n = 0$ も許し、このときは [ ] という空の型代入となる。

定義 15 [型代入の適用] 型代入 Θ が型変数 α を A に対応付けるとき (つまり，$\Theta = [\cdots, \alpha := A, \cdots]$ の形であるとき，$\Theta(\alpha) = A$ と定義し，α が Θ で対応付けられていないとき，$\Theta(\alpha) = \alpha$ と定義する．

この定義を，一般の型に対して以下のように拡張する．

$$\Theta(\texttt{int}) \stackrel{\text{def}}{=} \texttt{int}$$
$$\Theta(\texttt{bool}) \stackrel{\text{def}}{=} \texttt{bool}$$
$$\Theta(A \to B) \stackrel{\text{def}}{=} \Theta(A) \to \Theta(B)$$
$$\Theta(A \times B) \stackrel{\text{def}}{=} \Theta(A) \times \Theta(B)$$

型環境 $\Gamma = x_1 : A_1, \cdots, x_n : A_n$ に対しては、

$$\Theta(\Gamma) \stackrel{\text{def}}{=} x_1 : (\Theta(A_1)), \cdots, x_n : (\Theta(A_n))$$

と定義する．

たとえば，$\Gamma = x : \alpha \to \beta,\ y : \beta \times \gamma$ および $\Theta = [\alpha := \texttt{int} \to \delta,\ \beta := \texttt{bool}]$ のとき，$\Theta(\Gamma) = x : (\texttt{int} \to \delta) \to \texttt{bool},\ y : \texttt{bool} \times \gamma$ である．

7.3 代表的な型

先に述べたように、項 $\lambda x.\ (\texttt{left}(x), \texttt{right}(x))$ は $(\texttt{int} \times \texttt{bool}) \to (\texttt{bool} \times \texttt{int})$ や $(\texttt{int} \times \texttt{int}) \to (\texttt{int} \times \texttt{int})$ という型を持つ。また、型変数を含む型を使うと、$(\alpha \times \beta) \to (\beta \times \alpha)$ や $(\beta \times \beta) \to (\beta \times \beta)$ も、上記の項の型となっている。

これらの型のうち、$(\alpha \times \beta) \to (\beta \times \alpha)$ が代表的な型、あるいは、最も「良い」型であると言える。なぜなら、以下のように、この型の α, β に適当な型を代入すると、他の全ての型が得られるからである。

$$\begin{aligned}
[\alpha := \texttt{int},\ \beta := \texttt{bool}]((\alpha \times \beta) \to (\beta \times \alpha)) &= (\texttt{int} \times \texttt{bool}) \to (\texttt{bool} \times \texttt{int}) \\
[\alpha := \texttt{int},\ \beta := \texttt{int}]((\alpha \times \beta) \to (\beta \times \alpha)) &= (\texttt{int} \times \texttt{int}) \to (\texttt{int} \times \texttt{int}) \\
[\alpha := \beta]((\alpha \times \beta) \to (\beta \times \alpha)) &= (\beta \times \beta) \to (\beta \times \beta)
\end{aligned}$$

型推論アルゴリズムにおいて我々が求めたいのは、この「代表的な型」、つまり、$(\alpha \times \beta) \to (\beta \times \alpha)$ のような型であり、それに型代入を適用すると、他のすべての型が得られるものである。

7.4 型推論問題の定式化

ここまでの準備をもとに、型推論問題を定式化することにする。

型推論問題の定式化:

- 型文脈 Γ と (型がつかないかもしれない) 項 M に対して、$\Theta(\Gamma) \vdash M : A$ が導出できる Θ と A が存在するかどうかを判定する。(それが導出できる Θ と A のセット (Θ, A) を、Γ, M に対する型付けと呼ぶことにする[a])。
- そのような型付けが存在する場合、そのような Θ と A の中で代表的 (Principal) である Θ_0 と A_0 を求める。

[a] ここでは、(Θ, A) を「型付け」と呼んだが、これは便宜上のものである。通常の教科書では、「型付け」という言葉は、与えられた M に対して $\Gamma \vdash M : A$ が成立する Γ と A のセットのことを意味する。

ここで、Γ と M に対して、Θ_0 と A_0 が代表的な型付けであるとは、以下の条件が両方とも成立することである。

- $\Theta_0(\Gamma) \vdash M : A_0$ が導出できる.
- $\Theta(\Gamma) \vdash M : A$ が導出できるような任意の Θ, A に対して、ある型代入 Φ があって、$\Phi(\Theta_0(\Gamma)) = \Theta(\Gamma)$ および $\Phi(A_0) = A$ が成立する。

後半の条件は若干わかりにくかもしれないが、「Θ_0, A_0 に適当な型代入を適用すると、他のすべての型付け Θ, A が得られる」ということであり、要するに、Θ_0, A_0 が最も一般的、つまり、代表的であるということである。

いくつか例を述べる。

例 18　$\Gamma = x : \alpha \to \beta,\ y : \gamma$ および $M = \texttt{if}\ y\ \texttt{then}\ x@10\ \texttt{else}\ 20$ に対して、$\Theta = [\alpha := \texttt{int},\ \beta := \texttt{int},\ \gamma := \texttt{bool}]$, $A = \texttt{int}$ とすると、(Θ, A) は、その型付けの 1 つである。実際に、$\Theta(\Gamma) = x : \texttt{int} \to \texttt{int},\ y : \texttt{bool}$ であり、$x : \texttt{int} \to \texttt{int},\ y : \texttt{bool} \vdash \texttt{if}\ y\ \texttt{then}\ x@10\ \texttt{else}\ 20 : \texttt{int}$ が導出可能である。

なお,(Θ, A) は、上記の Γ, M に対する代表的な型付けである。

例 19　$\Gamma = x : \alpha,\ y : \beta$ および $M = x@(x@y)$ に対して、$\Theta_1 = [\alpha := \texttt{int} \to \texttt{int},\ \beta := \texttt{int}]$, $A_1 = \texttt{int}$ とすると、$(\Theta_1,\ A_1)$ は、その型付けの 1 つである。

また、$\Theta_2 = [\alpha := \texttt{bool} \to \texttt{bool},\ \beta := \texttt{bool}]$, $A_2 = \texttt{bool}$ とすると、$(\Theta_2,\ A_2)$ は、その型付けの 1 つである。

さらに、$\Theta_3 = [\alpha := \beta \to \beta]$, $A_2 = \beta$ とすると、$(\Theta_3,\ A_3)$ は、その型付けの 1 つであるとともに、代表的な型付けである。

例 20　$\Gamma = x : \alpha$ および $M = (\texttt{right}(x), \texttt{left}(x))$ に対して、$\Theta_1 = [\alpha := \texttt{int} \times \texttt{bool}]$, $A_1 = \texttt{bool} \times \texttt{int}$ とすると、$(\Theta_1,\ A_1)$ は、その型付けの 1 つである。

また、$\Theta_2 = [\alpha := \beta \times \beta]$, $A_2 = \beta \times \beta$ とすると、$(\Theta_2,\ A_2)$ は、その型付けの 1 つである。

さらに、$\Theta_3 = [\alpha := \gamma \times \delta]$, $A_3 = \delta \times \gamma$ とすると、$(\Theta_3,\ A_3)$ は、その型付けの 1 つである。これは、代表

的なな型付けである。

例 21　$\Gamma = x : \alpha,\ y : \beta$ および $M = (x@y) + (y@x)$ に対する型付けは存在しない。

この節の最後に、Γ と M の型付けについての定理を、証明なしで紹介する。

定理 12（型推論問題の決定可能性） 本資料の単純型付きラムダ計算の体系、計算体系 CoreML、および、後述する多相型の体系では、型文脈 Γ と項 M に対する型付けが存在するかどうかは決定可能である。(それを有限時間で判定するアルゴリズムが存在する。)

定理 13（Principal Type の存在） 本資料の単純型付きラムダ計算の体系、計算体系 CoreML、および、後述する多相型の体系では、型文脈 Γ と項 M に対する型付けが存在する場合、その代表的な型付けが存在する。

参考。ここまで述べてきた Principal Type は、「型推論問題その 2」に対応しており、Γ と M だけが与えられた時、$\Gamma \vdash M : A$ となる A のうち、代表的なもの (Principal な型) を意味している。一方で、「型推論問題その 1」に対応し、M だけが与えられた時、$\Gamma \vdash M : A$ となる Γ と A のうち代表的なものを考えることもある。これは「Γ と A のセット」であり、代表的な型付け (Principal な Typing) と呼ばれる。

多相型の体系以外の 2 つについては、型付け可能であれば、Principal Type も Principal Typing も存在する。一方、後で述べる多相型の体系では、型付け可能であれば、Principal Type は存在するが、Principal Typing は必ずしも存在しない (一意的な「代表」を選ぶことができなくなる)。

このような背景のもと、本講義資料では、簡単のため、Principal Type のみを解説した。

7.5　型推論アルゴリズムの概要

計算体系 CoreML に対して、前節の型推論問題を具体的に解くアルゴリズムを与えよう。

その概要は以下の通りである。

型推論アルゴリズム: 型文脈 Γ と (型がつかないかもしれない) 項 M を入力とし、「型付けできる/型付けできない」という情報と、型付できる場合は、その代表的な型付け Θ と A を出力する。

- ステップ 1. Γ と M に対して、型推論図を下から構成し、型に関する制約を生成する。
- ステップ 2. この制約を単一化アルゴリズムにかけて、制約を解消し、答えを得る。

ここで、型に関する制約 (constraint) とは、型に対する等式の集合であり、具体例は後述する。

7.6　型推論のステップ 1: 制約生成

制約生成アルゴリズムは、型付け図を下から構成する手順に従うだけである。一般的に書くほどの複雑さはないので、ここでは、具体的な例をあげる。

例 22　型環境 $\Gamma = y : \alpha$ と、項 $M = \lambda f.\lambda x.\ f@(x + y)$ が与えられたとする。

- フレッシュな型変数 (他で使われていない型変数) を 1 つ選ぶ。ここでは β とする。
- $\Gamma \vdash M : \beta$ の型導出の最後 (一番下) で使われた規則は lambda ルールなので、以下の形である。

$$\frac{y : \alpha,\ f : \square_1 \vdash \lambda x.\ f@(x + y) : \square_2}{y : \alpha \vdash M : \beta}\ \text{lambda}$$

ここで $\square_1, \square_2$ には何らかの型がはいるが、まだわからないので、フレッシュな型変数を 2 つ選びそこ

にいれる。ここでは、γ と δ とする。これにより、

$$\frac{y:\alpha,\ f:\gamma \vdash \lambda x.\ f@(x+y)\ :\ \delta}{y:\alpha \vdash M:\beta}\ \text{lambda}$$

となる。また、これが lambda ルールの正しい適用となるためには、$\beta = \gamma \to \delta$ でなければならない。そこで $\beta = \gamma \to \delta$ を、制約として残す。

- 次に、$y:\alpha,\ f:\gamma \vdash \lambda x.\ f@(x+y)\ :\ \delta$ を導いた最後の規則は lambda ルールなので、以下の形である。この場合も、わからない型が 2 つ出現するので、型変数を 2 つ選び、ϵ, ϕ とする。

$$\frac{y:\alpha,\ f:\gamma,\ x:\epsilon \vdash f@(x+y)\ :\ \phi}{y:\alpha,\ f:\gamma \vdash \lambda x.\ f@(x+y)\ :\ \delta}\ \text{lambda}$$

  $\delta = \epsilon \to \phi$ を制約に加える。
- 次に、最後に使った規則は apply ルールなので、以下の形である。この場合は、わからない型が 1 つなので、型変数を 1 つ選び ρ とする。

$$\frac{y:\alpha,\ f:\gamma,\ x:\epsilon \vdash f\ :\ \rho \to \phi \quad y:\alpha,\ f:\gamma,\ x:\epsilon \vdash x+y\ :\ \rho}{y:\alpha,\ f:\gamma,\ x:\epsilon \vdash f@(x+y)\ :\ \phi}\ \text{apply}$$

  ここでは、制約に追加するものはない。
- 次に、$\cdots \vdash f:\rho \to \phi$ の導出の最後に使った規則は var ルールである。この場合、新しい型変数の生成はない。$(f:\rho \to \phi) \in (y:\alpha,\ f:\gamma,\ x:\epsilon)$ となるためには、$\rho \to \phi = \gamma$ が必要であるので、これを制約に加える。
- 次に、$\cdots \vdash x+y:\rho$ の導出の最後に使った規則は plus ルールである。

$$\frac{y:\alpha,\ f:\gamma,\ x:\epsilon \vdash x\ :\ \texttt{int} \quad y:\alpha,\ f:\gamma,\ x:\epsilon \vdash y\ :\ \texttt{int}}{y:\alpha,\ f:\gamma,\ x:\epsilon \vdash x+y\ :\ \rho}\ \text{plus}$$

  この部分が正しい型付けとなるためには、$\rho = \texttt{int}$ が必要であるので、これを制約に加える。
- 残りは、x と y の部分であり、f の時と同様に考えると、$\epsilon = \texttt{int}$ および $\alpha = \texttt{int}$ が必要となり、これらを制約に加える。
- 以上で、M の型付け図は完成である。ここまでで生成された制約の集合は、以下の通りである。

$$\begin{aligned}\beta &= \gamma \to \delta\\ \delta &= \epsilon \to \phi\\ \rho \to \phi &= \gamma\\ \rho &= \texttt{int}\\ \epsilon &= \texttt{int}\\ \alpha &= \texttt{int}\end{aligned}$$

  作り方から明らかなように、これらの等式すべてが成立することと、$\Gamma \vdash M:\beta$ が導出できることは同値である。

この最後の部分で「等式が成立する」という言葉を、きちんと考えよう。型変数 $\alpha, \beta, \gamma, \delta, \rho, \phi, \epsilon$ というのは、もともとの型推論問題には含まれていないものであり、型推論の途中で勝手に導入したものである。それらは、「後で適当な型におきかえられるための穴」を表している。

よって、「等式が成立する」ということを正確に書きなおすと、「これらの型変数に適当な代入を適用したあと、等式が成立する」という意味である。この例のようにして、型推論問題は、「型に関する等式群を成立させることのできる型代入が存在するかどうか」という問題に帰着される。

7.7 型推論のステップ 2: 制約解消 (単一化)

ステップ 1 で生成された型制約を解くアルゴリズムについて述べる。型制約は、型変数を未知変数とする連立方程式であり、その解は、型変数がどういう型になっているかを示すもの、つまり、型代入として表される。

前節の例では以下の等式の集合 E が生成された。ただし、ここでは「まだ解かれていない方程式」という側面を強調するため、等式の「$=$」という記号を「$\stackrel{?}{=}$」に置きかえている。

$$E = \{\beta \stackrel{?}{=} \gamma \to \delta,\ \delta \stackrel{?}{=} \epsilon \to \phi,\ \rho \to \phi \stackrel{?}{=} \gamma,\ \rho \stackrel{?}{=} \mathtt{int},\ \epsilon \stackrel{?}{=} \mathtt{int},\ \alpha \stackrel{?}{=} \mathtt{int}\}$$

この等式集合の解の 1 つは、次の型代入 Θ である。

$$\begin{aligned}\Theta = [&\alpha := \mathtt{int},\ \beta := (\mathtt{int} \to \phi) \to (\mathtt{int} \to \phi),\\ &\gamma := \mathtt{int} \to \phi,\ \delta := \mathtt{int} \to \phi,\ \epsilon := \mathtt{int},\ \rho := \mathtt{int}]\end{aligned}$$

この Θ が、確かに上記の型制約の解になっていることを確かめるには、以下の等式が実際に成立していること (左右両辺の型が完全に同一の表現であること) を確かめればよい。

$$\begin{aligned}\Theta(\beta) &= \Theta(\gamma \to \delta)\\ \Theta(\delta) &= \Theta(\epsilon \to \phi)\\ \Theta(\rho \to \phi) &= \Theta(\gamma)\\ \Theta(\rho) &= \Theta(\mathtt{int})\\ \Theta(\epsilon) &= \Theta(\mathtt{int})\\ \Theta(\alpha) &= \Theta(\mathtt{int})\end{aligned}$$

一般に、型制約 (型に関する等式の集合) $\{A_1 \stackrel{?}{=} B_1, \cdots, A_n \stackrel{?}{=} B_n\}$ と型代入 Θ が、$\Theta(A_1) = \Theta(B_1), \cdots, \Theta(A_n) = \Theta(B_n)$ を満たす時 (これらの等式の左右両辺がそれぞれ同一の型となる時)、Θ はこの型制約の解 (の 1 つ) と言う。型制約に対して、その解が存在するかどうかを問う問題を「単一化問題 (unification problem)」と言う。

Robinson が考案した単一化アルゴリズム (unification algorithm) は、制約を満たす解 (より正確には、「制約を満たす解のうち代表的なもの」) を具体的に求めるアルゴリズムであり、その概要は以下の通りである。

unify(E, Θ): E は型制約 (型に関する等式の有限集合)、Θ は型代入で、型代入を返す。

(1) $E = \{\}$ のとき、「Θ という解あり」を答として返す。
(2) $E \neq \{\}$ のとき、E から任意の等式 1 つを選び $A \stackrel{?}{=} B$ とする。$E_1 = E - \{A \stackrel{?}{=} B\}$ とおく。
  (2-1) $A = B$ の時、つまり、もともと A と B が同一の型の時、unify(E_1, Θ) を呼び出して、その答を返す。
  (2-2) $A \neq B$ で、A が型変数のとき、
    (2-2-1) 型 B に A が現れるなら、「解なし」を答として返す。
    (2-2-2) 型 B に A が現れないなら、$\Theta_1 = [A := B]$ と置いた上で、unify$(\Theta_1(E_1),\ \Theta_1(\Theta))$ を計算し、その答を返す。
  (2-3) $A \neq B$ で、B が型変数のとき、A と B を逆にして 1 つ前のケースを適用。
  (2-4) $A \neq B$ で、$A = A_1 \times A_2$, $B = B_1 \times B_2$ のとき、$E_2 = E_1 \cup \{A_1 = B_1,\ A_2 = B_2\}$ と置き、unify(E_2, Θ) を呼び出し、その答を返す。
  (2-5) $A \neq B$ で、$A = A_1 \to A_2$, $B = B_1 \to B_2$ のとき、1 つ前のケースと同様。
  (2-6) 上記のいずれのケースでもないとき、「解なし」を答として返す。

このアルゴリズムで、やや難しいので、以下ではポイントを解説する。

まず、unify の第二引数 Θ は、「作りかけの答 (型代入)」を保持するためのものである。unify を最初に呼び出す時は、第二引数は空の型代入 [ ] とするが、再帰呼び出しにおいて、次第に大きな型代入となり、単一化が終了する時には答として返される。

上記のアルゴリズムにおける「$A \neq B$ で、A が型変数で、B に A が現れる」ケースは、たとえば、$A = \alpha$, $B = \beta \times \alpha \to$ int という場合である。この場合、α と β に何を代入しても、A と B が一致することはないので、「解なし」という答を返す[*18]。

「$A \neq B$ で、A が型変数で、B に A が現れない」ケースは、A は型変数なのでこれを α とおくと、型代入 $[\alpha := B]$ をすることにより、A と B が一致する。このとき、残りの等式 E_1 に含まれる α には、B を代入 (上記のアルゴリズムで、$\Theta_1(E_1)$ と表されている) すると共に、「作りかけの解」Θ に、$[\alpha := B]$ を追加 ($\Theta_1(\Theta)$ と表されている) する必要がある。

ここで、等式集合 $E = \{A_1 \stackrel{?}{=} B_1, \cdots, A_n \stackrel{?}{=} B_n\}$ に型代入 Θ_1 を適用した結果を、

$$\Theta_1(E_1) \stackrel{\text{def}}{=} \{\Theta_1(A_1) \stackrel{?}{=} \Theta_1(B_1),\ \cdots,\ \Theta_1(A_n) \stackrel{?}{=} \Theta_1(B_n)\}$$

と定義する。また、型代入 $\Theta_0 = [\alpha_1 := A_1, \cdots, \alpha_n := A_n]$ に、型代入 $\Theta_1 = [\beta_1 := B_1, \cdots, \beta_m := B_m]$ を適用した結果を、

$$\Theta_1(\Theta_0) \stackrel{\text{def}}{=} [\alpha_1 := \Theta_1(A_1), \cdots, \alpha_n := \Theta_1(A_n), \beta_1 := B_1, \cdots, \beta_m := B_m]$$

と定義する。たとえば、$[\gamma :=$ int$\times\delta]$ を $[\alpha :=$ bool $\to \gamma, \beta := \delta]$ に適用すると、$[\alpha :=$ bool $\to$ (int$\times\delta$), $\beta := \delta, \gamma :=$ int $\times\ \delta]$ を得る[*19].

[*18] 変数が式に現れるかどうかのチェックを、occurs check と言う。

[*19] この定義は、Θ_0 と Θ_1 が、同じ型変数に代入しない場合のみ有効である。たとえば、$[\alpha :=$ int$]$ を $[\alpha :=$ bool$, \beta :=$ int$]$ に適用すると、上記の定義では、$[\alpha :=$ bool$, \beta :=$ int$, \alpha :=$ int$]$ となるが、これは、α が二度現れてしまい、型代入とならない。なお、unify のアルゴリズムでは、そのようなケースは出てこない。

上記の単一化アルゴリズムは非決定的であり、E が 2 つ以上の等式を含む場合、どの等式から解消していくかは一意的ではない。しかし、実際には、どのような順番で等式を解消していっても、必ず停止し、かつ、その解は実質的に同じであることが証明されている。

例 23（単一化アルゴリズムの実行例 1） $E = \{\alpha \to \beta \stackrel{?}{=} \gamma \to \delta,\ \beta \stackrel{?}{=} \epsilon \times \mathtt{int},\ \delta \stackrel{?}{=} \epsilon \times \alpha\}$ に対して、$\mathtt{unify}(E, [\,])$ を実行する。(最初は、$\Theta = [\,]$ である。)

- E から $\alpha \to \beta \stackrel{?}{=} \gamma \to \delta$ を選ぶ。つまり、$A = \alpha \to \beta$, $B = \gamma \to \delta$ である。(2-5) に該当するので、次の unify の呼び出しでは、$E = \{\alpha \stackrel{?}{=} \gamma,\ \beta \stackrel{?}{=} \delta,\ \beta \stackrel{?}{=} \epsilon \times \mathtt{int},\ \delta \stackrel{?}{=} \epsilon \times \alpha\}$ および $\Theta = [\,]$ となる。
- E から $\alpha \stackrel{?}{=} \gamma$ を選ぶと (2-2-2) に該当する。次の unify の呼び出しでは、$E = \{\beta \stackrel{?}{=} \delta,\ \beta \stackrel{?}{=} \epsilon \times \mathtt{int},\ \delta \stackrel{?}{=} \epsilon \times \gamma\}$ および $\Theta = [\alpha := \gamma]$ となる。
- E から $\beta \stackrel{?}{=} \delta$ を選ぶと (2-2-2) に該当する。次の unify の呼び出しでは、$E = \{\delta \stackrel{?}{=} \epsilon \times \mathtt{int},\ \delta \stackrel{?}{=} \epsilon \times \gamma\}$ および $\Theta = [\alpha := \gamma,\ \beta := \delta]$ となる。
- E から $\delta \stackrel{?}{=} \epsilon \times \mathtt{int}$ を選ぶと (2-2-2) に該当する。次の unify の呼び出しでは、$E = \{\epsilon \times \mathtt{int} \stackrel{?}{=} \epsilon \times \gamma\}$ および $\Theta = [\alpha := \gamma,\ \beta := \epsilon \times \mathtt{int},\ \delta := \epsilon \times \mathtt{int}]$ となる。
- E から $\epsilon \times \mathtt{int} \stackrel{?}{=} \epsilon \times \gamma$ を選ぶと (2-4) に該当する。次の unify の呼び出しでは、$E = \{\epsilon \stackrel{?}{=} \epsilon,\ \mathtt{int} \stackrel{?}{=} \gamma\}$ および $\Theta = [\alpha := \gamma,\ \beta := \epsilon \times \mathtt{int},\ \delta := \epsilon \times \mathtt{int}]$ となる。
- E から $\epsilon \stackrel{?}{=} \epsilon$ を選ぶと (2-1) に該当する。次の unify の呼び出しでは、$E = \{\mathtt{int} \stackrel{?}{=} \gamma\}$ および $\Theta = [\alpha := \gamma,\ \beta := \epsilon \times \mathtt{int},\ \delta := \epsilon \times \mathtt{int}]$ となる。
- E から $\mathtt{int} \stackrel{?}{=} \gamma$ を選ぶと (2-3) に該当する。次の unify の呼び出しでは、$E = \{\}$ および $\Theta = [\alpha := \mathtt{int},\ \beta := \epsilon \times \mathtt{int},\ \delta := \epsilon \times \mathtt{int},\ \gamma := \mathtt{int}]$ となる。
- $E = \{\ \}$ なので、$\Theta = [\alpha := \mathtt{int},\ \beta := \epsilon \times \mathtt{int},\ \delta := \epsilon \times \mathtt{int},\ \gamma := \mathtt{int}]$ を返す。

例 24（単一化アルゴリズムの実行例 2） $E = \{\alpha \times \mathtt{bool} \stackrel{?}{=} \mathtt{int} \times \beta,\ \beta \stackrel{?}{=} \alpha \to \mathtt{int}\}$ に対して、$\mathtt{unify}(E, [\,])$ を実行する。最初は、$\Theta = [\,]$ である。

- E から $\alpha \times \mathtt{bool} \stackrel{?}{=} \mathtt{int} \times \beta$ を選ぶと (2-4) に該当し、次の unify の呼び出しでは、$E = \{\alpha \stackrel{?}{=} \mathtt{int},\ \mathtt{bool} \stackrel{?}{=} \beta,\ \beta \stackrel{?}{=} \alpha \to \mathtt{int}\}$ および $\Theta = [\,]$ となる。
- $\alpha \stackrel{?}{=} \mathtt{int}$ を選ぶと (2-2-2) に該当し、次の unify の呼び出しでは、$E = \{\mathtt{bool} \stackrel{?}{=} \beta,\ \beta \stackrel{?}{=} \mathtt{int} \to \mathtt{int}\}$ および $\Theta = [\alpha := \mathtt{int}]$ となる。
- $\mathtt{bool} \stackrel{?}{=} \beta$ を選ぶと (2-3) に該当し、次の unify の呼び出しでは、$E = \{\mathtt{bool} \stackrel{?}{=} \mathtt{int} \to \mathtt{int}\}$ および $\Theta = [\alpha := \mathtt{int},\ \beta := \mathtt{bool}]$ となる。
- $\mathtt{bool} \stackrel{?}{=} \mathtt{int} \to \mathtt{int}$ を選ぶと (2-6) に該当し、「解なし」が答となる。

例 25（型推論アルゴリズムの実行例） 型環境 $\Gamma = y : \alpha$ と、項 $M = \lambda f.\lambda x.\ f@(x + y)$ に対する型推論を行なう。

- ステップ 1 を実行する。これは、例 22 の通りであり、M の型を β と置いた上で、型制約 $E = \{\beta \stackrel{?}{=} \gamma \to \delta, \delta \stackrel{?}{=} \epsilon \to \phi, \rho \to \phi \stackrel{?}{=} \gamma, \rho \stackrel{?}{=} \mathtt{int}, \epsilon \stackrel{?}{=} \mathtt{int}, \alpha \stackrel{?}{=} \mathtt{int}\}$ が得られる。
- ステップ 2 を実行する。$\mathtt{unify}(E, [\,])$ の形で計算を行なうことにより、型代入 $[\alpha := \mathtt{int}, \beta := (\mathtt{int} \to \phi) \to (\mathtt{int} \to \phi), \delta := \mathtt{int} \to \phi, \gamma := \mathtt{int} \to \phi, \epsilon := \mathtt{int}, \rho := \mathtt{int}]$ が得られる。この型代入を Θ と置く。

- 型代入 Θ に対して、$\Theta(\Gamma) = y : \texttt{int}$ および $\Theta(\beta) = (\texttt{int} \to \phi) \to (\texttt{int} \to \phi)$ なので、

$$y : \texttt{int} \vdash M : (\texttt{int} \to \phi) \to (\texttt{int} \to \phi)$$

  が代表的な型付けであることがわかった。

問題 1　以下の項の型推論を行いなさい。(型推論に失敗する例も含まれていることに注意せよ。)

- $\lambda x.\lambda y.\ \texttt{if}\ x = 0\ \texttt{then}\ y\ \texttt{else}\ y + 1$
- $\lambda x.\ x@x$
- $\lambda x.\lambda y.\lambda z.\ (x@z)@(y@z)$
- $\lambda f.\lambda x.\ f@(f@x)$
- $\lambda x.\lambda y.\ (x@y) + (y@x)$
- $\lambda x.\ \texttt{fix}\ f.y.\texttt{if}\ y = 0\ \texttt{then}\ x\ \texttt{else}\ f@(y-1) + 1$
- $\lambda x.\ \texttt{fix}\ f.y.\texttt{if}\ y = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (f@(y-1))$

この章の最後に、単一化問題と単一化アルゴリズムに関する定理を証明なしで紹介する。

定理 14（単一化問題の決定可能性）　1. 本資料の単純型付きラムダ計算の体系、計算体系 CoreML、および、後述する多相型の体系に対して、型制約の単一化問題は決定可能である。(それを有限時間で判定するアルゴリズムが存在する。)

2. 型制約 E の単一化問題の解が存在すれば、E の代表的な解が存在する。ただし、代表的な解とは、適当な型代入を適用することによって、他のどんな解も得られるものである。

定理 15（単一化アルゴリズムの正しさ）　任意の型制約 E に対して、上記の `unify` は、必ず有限時間で停止し、もし、E の単一化問題の解が存在すれば、その代表的な解 (の 1 つ) を返し、解が存在しなければ「解なし」という答を返す。

前に述べた型制約の生成とあわせると、以下の性質が得られる。

定理 16（型推論アルゴリズムの正しさ）　本章で述べたアルゴリズムは、計算体系 CoreML の型推論問題に関して健全かつ完全である。

すなわち、Γ と M が与えられたとき、本章で述べたアルゴリズムは必ず停止し、型付け Θ と A が存在するならば、その代表的な型付けを返す。また、型付け Θ と A が存在するならば、型付けが存在しない旨を返す。

8　多相型の体系

型システムの役割は、7+true などの無意味な式を発見して、排除することにあった。しかし、単純な型システムでは、時として制限が強くなり過ぎてしまい、意味がある計算をあらわしている式まで排除してしまうことがある。そこで、型システムを拡張することが考えられる。

この章は、そのような拡張のうち、非常に強力で応用範囲の広い多相型 (polymorphic type) を扱う。これは、「すべての型」をあらわす型を導入して、型システムを拡張するものである。

まず、なぜ、そのような型が必要かを、例で見てみよう。

2 つの要素からなる対の左右をいれかえる関数 $\lambda x.\,(\mathtt{right}(x), \mathtt{left}(x))$ を型推論すると、$(\alpha \times \beta) \to (\beta \times \alpha)$ という型が得られる。よって、この関数を、$(\mathtt{int} \times \mathtt{int}) \to (\mathtt{int} \times \mathtt{int})$ という型の式として使うこともできるし、$(\mathtt{int} \times \mathtt{bool}) \to (\mathtt{bool} \times \mathtt{int})$ という型の式として使うこともできる。しかしながら、前節までの体系では、1 つのプログラムにおいて、ある場所では $(\mathtt{int} \times \mathtt{int}) \to (\mathtt{int} \times \mathtt{int})$ という型の式として使い、別の場所では、$(\mathtt{int} \times \mathtt{bool}) \to (\mathtt{bool} \times \mathtt{int})$ という型の式として使うことはできなかった。つまり、

$$\vdash \mathtt{let}\ f = \lambda x.\,(\mathtt{right}(x), \mathtt{left}(x))\ \mathtt{in}\ f@(10, 20)\ :\ \mathtt{int} \times \mathtt{int}$$
$$\vdash \mathtt{let}\ f = \lambda x.\,(\mathtt{right}(x), \mathtt{left}(x))\ \mathtt{in}\ f@(10, \mathtt{true})\ :\ \mathtt{int} \times \mathtt{bool}$$

は導出できる (これらの式は型を持つ) が、それらの両方を同時に使おうとして、

$$\mathtt{let}\ f = \lambda x.\,(\mathtt{right}(x), \mathtt{left}(x))\ \mathtt{in}\ (f@(10, 20),\ f@(10, \mathtt{true}))$$

とすると、$(\mathtt{int} \times \mathtt{int}) \times (\mathtt{int} \times \mathtt{bool})$ という型は付かずに、型付け不能になってしまう。

同様に、$\lambda x.x$ という式を、1 つのプログラムの中で、$\mathtt{int} \to \mathtt{int}$ と $\mathtt{bool} \to \mathtt{bool}$ の 2 つの目的で使うことはできなかった。つまり、

$$\mathtt{let}\ f = \lambda x.x\ \mathtt{in}\ \mathtt{if}\ f@\mathtt{true}\ \mathtt{then}\ (f@10) + 3\ \mathtt{else}\ 20$$

という式は型付けに失敗する。このようなプログラムを書きたい場合は、

$$\mathtt{let}\ f = \lambda x.x\ \mathtt{in}\ \mathtt{let}\ g = \lambda x.x\ \mathtt{in}\ \mathtt{if}\ f@\mathtt{true}\ \mathtt{then}\ (g@10) + 3\ \mathtt{else}\ 20$$

というように、型ごとに異なる変数に束縛して使い分ける必要があった。後者のプログラムは、前者よりも長くなり、実行時にプログラムが占めるメモリが無駄となるし、さらに、このようなプログラムの維持のコストが増大する (検証も大変になるし、プログラムを修正する場合に手間が増える) という問題が生じる。

もし、「任意の α と β に対して $(\alpha \times \beta) \to (\beta \times \alpha)$ 型である」や、「任意の α に対して $\alpha \to \alpha$ 型である」という事を意味する型を追加して、型の世界を拡張すれば、上記の式をいろいろな型に対して使うことができる。

多相型 (polymorphic type) は、このような要求に答えるものである。具体的には、$\forall\alpha.\forall\beta.((\alpha \times \beta) \to (\beta \times \alpha))$ や $\forall\alpha.\,(\alpha \to \alpha)$ などという表現を、型の世界に取りいれるものである。

8.1　多相型を持つ体系 CoreML$^+$

体系 CoreML に、このような多相型を取りいれた体系として、CoreML$^+$ を定義する。

まず、型の世界を以下のように拡張する。

$$A, B ::= \alpha \mid \mathtt{int} \mid \mathtt{bool} \mid A \times B \mid A \to B$$
$$P ::= A \mid \forall\alpha.P$$

A, B は CoreML の型と同じであるが、いま導入しようとしている多相型と区別して、単相型 (monomorphic type) と呼ぶ。P が多相型であり、その形は、単相型 A の前に 0 個以上の $\forall\alpha.$ をつけたものとなっている。この定義のものとで、たとえば、$\forall\alpha.\forall\beta.((\alpha \times \beta) \to (\beta \times \alpha))$ が型となることがわかるであろう。このほかにも、$\forall\alpha.((\alpha \times \texttt{bool}) \to (\texttt{int} \to \alpha))$ も (多相) 型となる。

2 種類の多相型: 上記の定義は、$\forall\alpha.$ が型の構文の一番外に現れることのみを許していて、内側に来ることは許していない。つまり、$(\forall\alpha.(\alpha \times \alpha)) \to \texttt{int}$ や、$\texttt{int} \times (\forall\alpha.\alpha)$ というものは、CoreML$^+$ における型ではない (単相型でも多相型でもない)。
この制限をせずに、$\forall\alpha.$ が型の内側に現れてもよいという定義にした体系は、System F と呼ばれ、Girard と Reynolds による多相型の体系として知られている。System F は、プログラミング言語の型システムの理論的基盤の 1 つとして有用であり広く使われているものであるが、現実のプログラミング言語で直接 System F の型をすべて許しているものはほとんどなく、ML でも許していない。その理由は、このような型を許すと、型推論ができなくなるという問題があるからである。ML の言語の趣旨は、人間が型を一切書かなくてもシステムが自動的に推論してくれる、ということなので、System F の型を全面的にいれるわけにはいかない。本節では、ML にならって、上記の制限をつけた多相型 (let 多相、あるいは ML 多相と呼ばれる) を導入した。
一方、プログラミング言語 Haskell では、System F の型 (および、それをさらに発展させた Fω と呼ばれる体系の型) を使うことができる。これらの型を使った式については、自動的に型推論できないので、プログラマが型を明示的に書く必要がある。

次に、CoreML$^+$ の型付け規則を述べる。まず判断は、$\Gamma \vdash M : A$ の形であるが、このうち Γ の形が以下のものに変更されている。

$$x_1 : P_1, \cdots, x_n : P_n$$

つまり、変数の型が CoreML では単相型に限定されていたが、CoreML$^+$ では多相型に拡張されている。なお、判断 $\Gamma \vdash M : A$ において M の型は単相型 A であることに注意されたい。

さて、CoreML$^+$ の型付け規則である。これは、驚くべきことに、ほとんどの規則が CoreML と同じであり (ただし、CoreML の型付け規則で Γ と書いてある部分は、上記のように「多相型を許す」という風に読みかえる必要がある)、変更が必要なのは、var 規則と let 規則の 2 つだけである。

まず、多相型を導入するのは、ML では let 式の役割としている。

$$\frac{\Gamma \vdash M : A \quad \Gamma, x : \texttt{Gen}(\Gamma; A) \vdash N : B}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : B}\ \text{let}$$

ここで、右側の前提における x の型が、CoreML では A だったものが、$\texttt{Gen}(\Gamma; A)$ になった点が変更点である。

これを定義するため、まず、「型に含まれる自由な型変数の集合」という概念を定義する。単相または多相

型 A, P に対して、その中に含まれる自由な型変数の集合 $\mathtt{FTV}(A)$ や $\mathtt{FTV}(P)$ は以下のように定義される。

$$\begin{aligned}
\mathtt{FTV}(\alpha) &\stackrel{\text{def}}{=} \{\alpha\} \\
\mathtt{FTV}(\mathtt{int}) &\stackrel{\text{def}}{=} \{\} \\
\mathtt{FTV}(\mathtt{bool}) &\stackrel{\text{def}}{=} \{\} \\
\mathtt{FTV}(A \to B) &\stackrel{\text{def}}{=} \mathtt{FTV}(A) \cup \mathtt{FTV}(B) \\
\mathtt{FTV}(A \times B) &\stackrel{\text{def}}{=} \mathtt{FTV}(A) \cup \mathtt{FTV}(B) \\
\mathtt{FTV}(\forall\alpha.P) &\stackrel{\text{def}}{=} \mathtt{FTV}(P) - \{\alpha\}
\end{aligned}$$

ここで、$-$ というのは差集合を取る演算であり、たとえば、$\{a, b\} - \{a, c\} = \{b\}$ である。自由な型変数の集合の例としては、$\mathtt{FTV}(\forall\alpha.(\alpha \to (\beta \times \mathtt{int}))) = \{\alpha, \beta\} - \{\alpha\} = \{\beta\}$ となる。

次に、Gen を定義する。

$$\begin{aligned}
\mathtt{Gen}(\Gamma; A) &\stackrel{\text{def}}{=} \forall\alpha_1.\cdots\forall\alpha_n.A \\
&\quad \text{where} \;\; \mathtt{FTV}(A) - \mathtt{FTV}(\Gamma) = \{\alpha_1, \cdots, \alpha_n\}
\end{aligned}$$

$\mathtt{Gen}(\Gamma; A)$ は、直感的には、A に含まれる型変数たちのうち、Γ に (自由に) 出現しないものを多相型にする ($\forall\alpha.$ をつける) ということである。

やや複雑な定義なので、例を見てみよう。

例 26　Gen の計算例をあげる。

$\mathtt{Gen}(x : \alpha \to \alpha,\ y : \mathtt{int} \times \beta;\ \alpha \times \beta \to \gamma) = \forall\gamma.(\alpha \times \beta \to \gamma)$

$\mathtt{Gen}(x : \forall\alpha.(\alpha \to \alpha),\ y : \mathtt{int} \times \beta;\ \alpha \times \beta \to \gamma) = \forall\alpha.\forall\gamma.(\alpha \times \beta \to \gamma)$

これらの定義を踏まえて let 規則を見てみよう。たとえば、$M = \lambda y.y$ のとき、$\vdash M : \alpha \to \alpha$ が推論できる。そこで、$x : \forall\alpha.(\alpha \to \alpha)$ という仮定のもとで N の型を推論してよい、ということであり、この x を多相型で使えることになる。

同様に、$M = \lambda y.(\mathtt{right}(y), \mathtt{left}(y))$ のとき、$\vdash M : (\alpha \times \beta) \to (\beta \times \alpha)$ が推論できる。そこで、$x : \forall\alpha.\forall\beta.((\alpha \times \beta) \to (\beta \times \alpha))$ という仮定のもとで N の型を推論してよい、ということであり、この x を多相型で使えることになる。

次に、変数規則において、多相型を使う。これは、以下の形にかわる。

$$\frac{((x : P) \in \Gamma) \quad (A \preceq P)}{\Gamma \vdash x : A}\ \mathtt{var}$$

ここで $A \preceq P$ は以下のように定義される。$P = \forall\alpha_1.\cdots\forall\alpha_n.B$ の形とするとき、ある型代入 Θ で、$\Theta(B) = A$ となるとき、および、その時にかぎり、$A \preceq P$ と定義する。

これも例を見てみよう。

$$\begin{aligned}
\mathtt{int} \times \beta \to \mathtt{int} &\preceq \forall\alpha.(\alpha \times \beta \to \alpha) \\
\mathtt{bool} \times \beta \to \mathtt{bool} &\preceq \forall\alpha.(\alpha \times \beta \to \alpha)
\end{aligned}$$

また、$\mathtt{int} \times \mathtt{int} \to \mathtt{int} \preceq \forall\alpha.(\alpha \times \beta \to \alpha)$ ではない。

var 規則は、変数 x が多相型をもつとき、x を使う場面では、その型変数を好きな型に具体化して使ってよいということである。

上記の 2 つの規則を使って、多相型を利用した式の型付けを見てみよう。

まず、$\lambda x.x$ を 2 つの異なる型 $\alpha \to \alpha$ と $(\alpha \to \alpha) \to (\alpha \to \alpha)$ で使う例である。

$$\dfrac{\dfrac{\overline{x : \beta \vdash x : \beta}}{\vdash \lambda x.x : \beta \to \beta} \quad \dfrac{\overline{f : \forall\beta.(\beta \to \beta) \vdash f : (\alpha \to \alpha) \to (\alpha \to \alpha)} \quad \overline{f : \forall\beta.(\beta \to \beta) \vdash f : \alpha \to \alpha}}{f : \forall\beta.(\beta \to \beta) \vdash f@f : \alpha \to \alpha}}{\vdash \texttt{let}\ f = \lambda x.x\ \texttt{in}\ f@f : \alpha \to \alpha}$$

この型付け図の左の方で、$\vdash \lambda x.x : \beta \to \beta$ となっていることから、$\texttt{Gen}(\ ; \beta \to \beta) = \forall\beta.(\beta \to \beta)$ を得る。右の方で、この f に対する var 規則が 2 回現れるが、それぞれで β を異なる型に具体化している。

次に $M = (\texttt{let}\ f = \lambda x.\ (\texttt{right}(x), \texttt{left}(x))\ \texttt{in}\ (f@(10, 20),\ f@(10, \texttt{true})))$ の型付けは以下の通りである。(ここで、$N = \lambda x.\ (\texttt{right}(x), \texttt{left}(x))$ および $P = \forall\alpha.\forall\beta.(\alpha \times \beta) \to (\beta \times \alpha)$ と置いた。)

$$\dfrac{\begin{array}{c}\vdots \\ \vdash N : (\alpha \times \beta) \to (\beta \times \alpha)\end{array} \quad \dfrac{\dfrac{\overline{f : P \vdash f : (\texttt{int} \times \texttt{int}) \to (\texttt{int} \times \texttt{int})}\ \cdots}{f : P \vdash f@(10, 20) : \texttt{int} \times \texttt{int}} \quad \dfrac{\overline{f : P \vdash f : (\texttt{int} \times \texttt{bool}) \to (\texttt{bool} \times \texttt{int})}\ \cdots}{f : P \vdash f@(10, \texttt{true}) : \texttt{bool} \times \texttt{int}}}{f : P \vdash (f@(10, 20),\ f@(10, \texttt{true})) : (\texttt{int} \times \texttt{int}) \times (\texttt{int} \times \texttt{bool})}}{\vdash M : (\texttt{int} \times \texttt{int}) \times (\texttt{int} \times \texttt{bool})}$$

この例でも、左の N の型付けにおいては、Γ の部分が空なので、$\texttt{Gen}(;\ (\alpha \times \beta) \to (\beta \times \alpha)) = \forall\alpha.\forall\beta.(\alpha \times \beta) \to (\beta \times \alpha) = P$ となる。右側の f に対する var 規則 (2 か所) で、この α, β を異なる型に具体化していることがわかる。

体系 CoreML$^+$ における計算規則は CoreML と同じであるので省略する。

以上のように、体系 CoreML$^+$ では、多相型を導入 (let 規則) し、多相型を使う (var 規則) ということを実現している。なお、CoreML$^+$ においては、λ で束縛される変数は多相型でないことを要求しているので (lambda 規則や apply 規則は、CoreML と同じであるため、単相型しか許されない)、λ を使って let をあらわすことはできない。すなわち、CoreML においては、$\texttt{let}\ x = M\ \texttt{in}\ N$ と $(\lambda x.N)\ @\ M$ は全く同じ (型付けるかどうか、と、計算した場合の意味の両方が同じ) であったが、CoreML$^+$ では型付けは同じではない。たとえば、$\texttt{let}\ f = \lambda x.x\ \texttt{in}\ f@f$ は上述した通り型付け可能であるが、$(\lambda f.\ f@f)\ @\ (\lambda x.x)$ は型付けできない。(後者は、f が単相型であるという制限のため型付けに失敗する。)

8.2 CoreML$^+$ に対する型推論

体系 CoreML$^+$ における型推論についても簡単に触れておこう。

ML 系の言語においては、System F のような多相型ではなく、制限された多相 (let 多相) を導入しているが、これは、このように制限すると、型推論ができる (型を持つかどうかが決定可能であり、型推論アルゴリズムが存在する) からである。

CoreML$^+$ に対する型推論アルゴリズムは、CoreML における型推論アルゴリズムの一点のみを修正すれば得られる。すなわち、「型推論のステップ 1: 制約生成」において、$\texttt{let}\ x = M\ \texttt{in}\ N$ の型推論図の作成を修正する。

これも、一般論ではなく例のみで見ることにする。型環境 Γ および項 $\texttt{let}\ f = \lambda x.x\ \texttt{in}\ f@f$ と型 A に対する制約生成を行なっているとする。($\Gamma \vdash \texttt{let}\ f = \lambda x.x\ \texttt{in}\ f@f : A$ という判断に対する制約生成を行っているとする。)

このとき、CoreML$^+$ の let 規則を適用するかわりに、まず項を、$\texttt{let}\ f_1 = \lambda x.x\ \texttt{in let}\ f_2 = \lambda x.x\ \texttt{in}\ f_1@f_2$ に変形して、CoreML の let 規則を何回か適用すればよい。すなわち、以下のようになる。

- 新しい型変数 α_1 を用意して、$\Gamma \vdash \lambda x.x : \alpha_1$ に対する制約生成を行なう。
- 次に、$\Gamma, f_1 : \alpha_1 \vdash \texttt{let}\ f_2 = \lambda x.x\ \texttt{in}\ f_1@f_2$ に対する制約生成を行なう。このために、新しい型変数 α_2 を用意して、$\Gamma \vdash \lambda x.x : \alpha_2$ に対する制約生成を行なう。
- 次に、$\Gamma, f_1 : \alpha_1, f_2 : \alpha_2 \vdash f_1@f_2$ に対する制約生成を行なう。

このようにすることにより、f が出現するたびに違う型になってよい、という多相性に対応した型推論アルゴリズムとすることができる。

一般に、f が let の本体で N 回でてきた場合は N 回の (単相型の)let の繰返しに置きかえて型推論を行えばよい。

このように変形した型推論アルゴリズムが、CoreML$^+$ の型推論問題に対して正しいことは、CoreML の場合と同様に示すことができる。

演習問題: 以下の式の型付け図を、体系 CoreML$^+$ で書きなさい。

1. 適当な A に対して、$\vdash \texttt{let}\ f = \lambda x.\lambda y.x\ \texttt{in}\ (f@\texttt{true})@(f@10@\texttt{false}) : A$
2. 適当な B に対して、$\vdash \texttt{let}\ f = \lambda x.(x, x)\ \texttt{in}\ (f@f, f@10) : B$

9　計算体系と論理体系の関係

本講義の主題は，プログラミング言語に関する研究と形式論理に関する研究が密接に関連することを学ぶことであった．本章では，いよいよ，計算体系と論理体系の関係を解き明かす．

9.1　型付きラムダ計算と直観主義命題論理の関係

これまでの演習を通して，型付きラムダ計算の体系と，直観主義命題論理の体系が極めて似ていることに気付いただろう．

たとえば，$A \to B$ という型は，$A \supset B$ という命題と非常によく似た規則を持っている．

- $\to$ の導入規則 (λ) と $\supset$ の導入規則 ($\supset I$)

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}\ \lambda \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ \supset I$$

- $\to$ の除去規則 ($apply$) と $\supset$ の除去規則 ($\supset E$)

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ apply \quad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \supset E$$

計算体系の $M : A$ という形から $M :$ を取り除けば論理体系の対応する規則になることがわかる．

同じことは，$\times$ の規則 ($pair, left, right$) と $\wedge$ の規則 ($\wedge I, \wedge EL, \wedge ER$)，$+$ の規則 ($inl, inr, case$) と $\vee$ の規則 ($\vee IL, \vee IR, \vee E$) についても言える．また，変数の規則 (var) は仮定を導入する規則 ($assume$) に対応する．全ての場合において，計算体系における $M : A$ という形から項の部分 $M :$ を取り除けば論理体系の対応する規則になることがわかる．

これは偶然であろうか？そうではない．次章で見るようにこれは非常に本質的な現象である．ここでは，その準備として，計算体系と論理体系に若干の (非本質的な) 変更を加える．

- 型付きラムダ計算の体系に，空の型 (要素を持たない型) として $\bot$ を導入する．
  $\bot$ は以下の規則 ($\bot$ 型の除去規則) だけを持つ型である．

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash abort(M) : A}\ abort$$

  直感的には，$\bot$ 型は「空の型」であり，要素を持たないものである．したがって，もし $\bot$ 型の値を計算する項 M があったとすれば，Γ 自体が矛盾した仮定であるので，そこからどんな型 A の要素も計算できてしまえる，というのがこの規則の意味である．このような型を導入しても，計算体系は本質的には何も変わらない，ということが直感的に想像されるであろう．すなわち，$abort$ を含む項 (プログラム) を実行したとき，$abort(M)$ という部分は決して実行されない (その部分に制御が行くことはない)．要素を持たない型は存在意義がないと考えるかもしれないが，Java 言語の try-catch, ML 言語の exception などで表現される例外機構の記述では有用である．それについては，「古典論理への拡張」の節で述べる．
- 直観主義命題論理の体系で $\neg$ は省略形と考える．すなわち，$\neg A$ は $A \supset \bot$ の省略形と見なし，$\neg$ という記号はないと考える．したがって，$\neg$ に関する 2 つの規則 $\neg I$ と $\neg E$ もないものと考える．

9.2 Curry(-Howard) の同型対応

Curry-Howard の同型対応は，「命題＝型」原理 (Propositions-as-Types Principle) とも呼ばれ，命題と型が本質的に同じであることを主張するものである．しかし，それは，単に命題と型が同じ導出規則で生成されるという静的な性質の対応だけではない．命題の導出と型の導出とが対応し，さらに，命題の導出に対する計算と型の導出に対する計算が対応する，という動的な性質の対応を含むものである．

ここで，「命題の導出に対する計算」とは何であろうか？それは前章で見た「無駄を省く変形操作」のことである．では，「型の導出に対する計算」とは何であろうか？型の導出とは，何らかの項がある型を持つ，ということの導出のことであり，「型つきラムダ項の導出」と呼んでいたものであった．つまり，「型の導出に対する計算」とは，型つきラムダ項の計算に他ならない．

以上まとめると以下の表を得る．

論理体系	計算体系
命題	型
$\supset$	$\rightarrow$
$\wedge$	$\times$
$\vee$	$+$
$\bot$	$\bot$
命題の導出 ($\vdash A$ の導出)	型の導出 ($\vdash A'$ の導出)
$\supset I$ 規則, $\supset E$ 規則	$lambda$ 規則, $apply$ 規則
$\wedge I$ 規則, $\wedge EL$ 規則 $\wedge ER$ 規則	$pair$ 規則, $left$ 規則, $right$ 規則
$\vee IL$ 規則, $\vee IR$ 規則, $\vee E$ 規則	inl 規則, inr 規則, $case$ 規則
$\bot E$ 規則	$abort$ 規則
この導出に対する変形操作 (無駄を省く変形)	この導出の計算 (項の計算)
$\supset I - \supset E$ 変形	β 計算規則
$\wedge I - \wedge EL$ 変形	pair-left 計算規則
$\wedge I - \wedge ER$ 変形	pair-right 計算規則
$\vee IL - \vee E$ 変形	case-inl 計算規則
$\vee IR - \vee E$ 変形	case-inr 計算規則

この表の意味するところは以下の通りである．

- 命題 A は，それにあらわれる $\supset, \wedge, \vee, \bot$ をそれぞれ $\rightarrow, \times, +, \bot$ に読みかえることにより型 A' に対応する．
- A の導出 (正確には，$\vdash A$ という判断の導出) は，その中に現れる規則を，表の 2 番目に列挙した対応関係 ($\supset I$ を λ 規則で置きかえ，$\supset E$ を $apply$ 規則で置きかえる，等) で置きかえることにより，A' の導出 (正確には，適当な項 M に対して $\vdash M : A'$ の導出) になる．
- A の導出に対する計算 (変形操作の系列) は，各変形操作を，表の 3 番目に列挙した対応関係 ($\supset I - \supset E$ 変形を，β 計算規則で置きかえる，等) で置きかえることにより，A' の導出に対する計算となる．

すなわち，直観主義論理の命題の導出と計算は，型付きラムダ計算の項の導出と計算に完全に対応する．これらは，片方が与えられれば他方が機械的に (一意的に) 定まる，という全単射で対応付けられている．ところで，計算体系にしても論理体系にしても，我々は形式的な構文と導出のみで定まるものとしてきたので，これらが完全に対応する，ということは，直観主義論理と型付きラムダ計算の体系は同じものである，ということになる．(もちろん，最初にこれらの体系を構築した際に意図した意味論は違うものであったはずだが，結果として形式化されたものはまったく同じである．)

これが Curry-Howard の同型対応である．Curry-Howard の同型対応は，計算と論理の形式的体系の間の深い関係を示すものである．なお，歴史的には，ここで述べたような命題論理の対応は Curry が発見し，後に少々触れる述語論理まで拡張した際の対応を Howard が発見したので，本節で述べた範囲での対応は正確には「Curry の同型対応」というべきである．

9.3 Curry-Howard の同型対応の周辺

前節の結果は理論的には美しい結果であるが，ただちにいくつかの疑問がわくだろう．

- そもそも何故 Curry-Howard の同型対応がなりたったのであろうか？
- Curry-Howard の同型対応が発見されたからといって，何が嬉しいのだろうか？ 2 つの独立に作られたものが同じことがわかると，何か本質的な進歩があるのだろうか？
- 型付きラムダ計算に対応するのは，古典論理ではなく，直観主義論理である．我々の日常的推論が古典論理でおこなわれていることを考えると，直観主義論理と対応することの意味は何だろうか？
- 形式的体系がたまたま同じだからといって，それは形式化のやり方に依存することであって，「計算」と「論理」が本質的に同じという結論を導いていいのだろうか？それぞれの意味も同じといっていいのだろうか？

これらの疑問について以下では考えてみる．

9.3.1 なぜ対応するのか？

直観主義論理の形式的体系に「ぴったり一致する」意味論として，構成的解釈があることを既に学んだ．これを使って，Curry-Howard 同型対応が「必然的」なものであることが説明できる．

- 構成的解釈において $A \supset B$ という命題の証拠は，「A の証拠をもらって B の証拠を返す関数」であると定められた．「命題 A の証拠」を「命題 A に対応する型 A' に族する要素」という対応関係でみると，「A の証拠をもらって B の証拠を返す関数」は「A 型の要素をもらって B 型の要素を返す関数」であり，そのような関数を集めた型は，ちょうど $A \to B$ という型になる．
- 構成的解釈において $A \wedge B$ の証拠は「A の証拠と B の証拠の対」であった．これを「命題 A の証拠」を「命題 A に対応する型 A' に族する要素」という対応関係でみると，「A' 型の要素と B' 型の要素の対」であり，これを集めた型は，ちょうど $A \times B$ という型になる．
- 構成的解釈において $A \vee B$ の証拠は，「A であるか B であるかをあらわす 1bit の情報と，A の証拠もしくは B の証拠の対」であった．これを「命題 A の証拠」を「命題 A に対応する型 A' に族する要素」という対応関係でみると，「A' 型であるか B' 型であるかを決める 1bit の情報」と，「A' 型の要素，もしくは，B' 型の要素」の対になる．これを集めた型は，ちょうど $A + B$ という型になる．

- 命題 $\bot$ には証拠がないが，これは，要素を持たない型 $\bot$ に対応している．

言い換えると，構成的解釈を表現するのに必要なだけの表現力をもった計算体系が型付きラムダ計算であるといえる．このように考えると，直観主義論理と型付きラムダ計算が対応するのは，むしろ当然のことと言える．

9.3.2 なにが嬉しいのか？

一般に，全く異なる生い立ちをもつ 2 つの理論の関係が解き明かされると，新たな発展が期待される．Curry-Howard の同型対応でも事情は同じであり，計算体系と論理体系で別々に知られていた事実の間の関係がわかったり，片方の結果を他方に移すことで新たな展開が知られたりする．このような観点を基礎とする研究は非常に盛んであり，**プログラミング言語論の研究の中心的テーマの 1 つ**である．

この章の後の方で，そのような話題を 4 つ紹介する．

9.3.3 直観主義論理と対応したことの意味は何か？

直観主義論理は，構成的解釈に対応する形式的体系であることからわかるように「計算」によって論理を解釈しようとする体系である．したがって，このような論理が計算体系と対応したからといって，あまり驚きはないかもしれない．いずれにせよ，**「直観主義論理＝計算の論理」**である．

では，純粋な論理として見たとき，直観主義論理はどんなものであろうか？既に見てきたように，本講義のような定式化をする限り，直観主義論理の証明は古典論理の証明よりはるかに自然である．命題 A が導出可能なとき，命題 A の部分命題しかあらわれないような導出が必ず存在する．このような論理は，純粋な論理体系としても非常に興味深いものであることが想像できよう．

さらに，近年の数学の流れとして，構成的な存在証明への関心があげられる．存在証明とは，$\exists x.A(x)$ という形の定理の証明のことである．一般に，数学の定理は排中律等を多用して証明されるため，$\exists x.A(x)$ が証明されたからといって，この x を具体的に計算することができる保証はまったくない．実際，「すべての実数を順番に並べる方法がある」という定理が成立するが，具体的な並べ方を紙に書くことは (どんな数学者といえども) できない．また，「どんな円でも半径と円周の長さの比率が一定である」という定理から円周率を計算することはできないだろう．しかし，近年の数学の傾向として，$\exists x.A(x)$ の証明は，構成的であることが望ましい，とされるようになってきた．構成的な存在証明とは，その証明の中に「x を具体的に計算する手続き」がはいっているものである．構成的証明は，直観主義論理で証明を作ることに相当し，一般の (非構成的かもしれない) 証明は，古典論理で証明を作ることに相当する．直観主義論理は，古典論理における排中律 (あるいはそれと同等な他の原理) を使うことができないので，証明を作る段階では苦労が多いが，証明できてしまえば，「計算手続きを含む」という非常に良い性質を持つものである．

以上のように，直観主義論理，あるいは，構成主義は，純粋な論理や数学の世界でも注目されている重要な論理であるといえる．

9.3.4 では，計算と論理は本当に同じか？

この問に対する答えは 1 つではない．論理体系や計算体系の定式化 (形式的体系を作ること) は，一意的ではなく，多様な形式的体系があり得るからである．

論理体系の形式化は，3 つのスタイルがある．

- 自然演繹 (natural deduction), 本講義で採用したスタイル

- シーケント計算 (sequent calculus)

$$\frac{}{\Gamma, A \vdash \Delta, A}\ intial \qquad \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \supset B}\ \supset R \qquad \frac{\Gamma_1, B \vdash \Delta_1 \quad \Gamma \vdash A, \Delta_2}{\Gamma_1, \Gamma_2, A \supset B \vdash \Delta_1, \Delta_2}\ \supset L$$

  自然演繹における I(導入規則) と E(除去規則) のかわりに，このように，L($\vdash$ の左側に導入) 規則と R($\vdash$ の右側に導入) 規則とがある．

- Hilbert 流 (Hilbert が考案したスタイル)

$$\frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B}\ modus\ ponens$$

$$\frac{}{\Gamma \vdash (A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))}\ S \qquad \frac{}{\Gamma \vdash A \supset (B \supset A)}\ K$$

  推論規則は modus ponens だけで，あとは全て S, K のような公理である．

古典論理や直観主義論理など多くの論理体系は，これら 3 つのスタイルのいずれでも形式化でき，どのスタイルでの形式化も同等 (証明できる論理式は同じ) であることが知られている．

Curry-Howard の同型対応は，このうち，自然演繹スタイルと Hilbert スタイルに対して成立するものである．シーケント体系スタイルに対しては Curry-Howard の同型対応に相当するものはない．したがって，シーケント計算スタイルでしか与えられていない論理体系は，対応する計算体系がない，ということになる．

一方，プログラミング言語は非常に複雑なものが多い (ほとんどの実用的なプログラミング言語は，その厳密な定義を書くと 1 冊の本になる．) 本講義で述べた型付きラムダ計算であれば，それに対応する論理体系を考えることは容易であったが，たとえば，C 言語に対応する論理体系を厳密に定義するのはほとんど不可能である．

このような観点から，計算と論理が「どんな場合にも完全に一致する」ということは言えないことがわかる．しかしながら，以下のようなことは言える．

- 論理体系のみ，あるいは，計算体系のみを考察対象としているときに比べて，Curry-Howard の同型対応が成立するような論理体系と計算体系がある場合，(後に述べる 4 つの例からもわかるように) 体系に関する種々の良い性質を証明する手だてが用意され，見通しの良い議論を展開することができる．
- したがって，論理体系のみ，あるいは，計算体系のみが与えられた場合，「Curry-Howard の同型対応が成立するような相棒はなにか？」を考えることは有意義である．そのような相棒が存在しない場合，多くの場合は，もとの体系が適切に構成されていないことを意味する．
- すなわち，C-H の同型対応が成立するように論理/計算体系を設計すべき，という設計の指針となる．

9.4 対応関係の拡張

この章では，Curry-Howard の同型対応を拡張することにより，プログラミング言語および論理体系に対する新たな知見が得られる例を 4 つ紹介する．本節で述べられるのは，どれもごく表面的な introduction のみであるので詳しくは大学院での授業「プログラム理論特論」や書籍等を参照されたい．

9.4.1 その 1: 限量子と依存型

この講義における論理は，命題論理であった．しかし，我々の日常の推論は (少なくとも) 一階述語論理程度の表現力を必要とする．論理体系を命題論理から一階述語論理に拡張したとき，対応する計算体系が型付きラムダ計算からどのようなものに拡張されるかを考えてみよう．

一階述語論理の論理記号は，命題論理のそれに比べて，$\forall$ や $\exists$ という 2 つの論理記号が増えている．

そこで，$\forall x.A(x)$ という命題を型と見なそう．この命題の証拠は，構成的解釈によると，「x をもらうと $A(x)$ の証拠を返す関数」である．これを型の世界の言葉で書くと「x をもらうと $A(x)$ という型の要素を関数」となる．このように，項 x に依存して決まる型 $A(x)$ を依存型 (dependent type) という．

例: `newlist` を，自然数 n をもらうと，長さ n のリスト (要素はすべて 0) を返す関数とする．

$$\texttt{newlist}(n) = \begin{cases} \text{空リスト} & (n = 0 \text{ の時}) \\ \texttt{cons}(0, \texttt{newlist}(n-1)) & (n > 0 \text{ の時}) \end{cases}$$

ここで $\texttt{cons}(a, L)$ は，リスト L の先頭に要素 a を追加したリストを表す．たとえば，$\texttt{newlist}(3) = \langle 0, 0, 0\rangle$ となることは想像できよう (ここで，リストを $\langle a, b, c\rangle$ のように表現した．)

このような関数はしばしば現れる有用な関数である．では，その型は何であろうか？「リスト」を表す型を `List` とすると，$\texttt{newlist} : \texttt{Nat} \to \texttt{List}$ となる．これは正しい型付けではあるが，応用局面によっては，より詳しい情報が欲しいことがある．つまり，「長さが不明なリスト」ではなく「長さ n のリスト」が返されることを表現したい場合である．そのような型を $\texttt{List}(n)$ とすると，$\texttt{newlist}: \texttt{Nat} \to \texttt{List}(n)$ という型を持たせたいが，これでは正しい型とはならない．(変数 n が自由に現れてしまっている．実際にはこの n は引数のことである．) そこで，このような型を，

$$\texttt{newlist} : \Pi(n : N)\texttt{List}(n)$$

と表現して，`newlist` が「自然数 n をもらうと，$\texttt{List}(n)$ 型の要素を返す関数」の型を持つことを意味する．この Pi が依存型を構成する型構成子の 1 つである．Π は Curry-Howard の同型対応のもとで $\forall$ に対応する．

一方，$\exists x.A(x)$ という命題の証拠は，「x と，$A(x)$ の証拠の対」である．これは Σ という型構成子で構成される型に相当する．たとえば，$< 3, (0\ 0\ 0) >$ が $\Sigma(n : N)List(n)$ の要素である．

以上まとめると，以下の対応関係になる．

直観主義の一階述語論理	依存型を持つ型付きラムダ計算
$\forall$	Π 型
$\exists$	Σ 型

この対応は，命題論理と (依存型を持たない) 型付きラムダ計算との対応 (Curry の対応) の拡張となっている．この $\forall$ や $\exists$ への拡張部分を Howard が示したので，全体を合わせて Curry-Howard の対応と言う．

依存型は，随分奇妙な型と思うかもしれないが，論理の世界では昔から知られた $\forall$ や $\exists$ のことだと思うとわかりやすい．プログラム言語の方での依存型の重要性は近年認識されてきており，後の応用局面で重要な意味を持つ．

9.4.2 その 2: 帰納と帰納型

前節の拡張により，一階述語論理 (の直観主義論理版) まで拡張されたが，これではまだ数学的な表現のためには力不足である．最も基礎的な数学は自然数に関する理論であり，そこでの証明は数学的帰納法が鍵となる．

$$\frac{\Gamma \vdash A(0) \quad \Gamma, A(x) \vdash A(x+1)}{\Gamma \vdash \forall x.(Nat(x) \supset A(x))} \text{ (数学的帰納法)}$$

このルールは，$A(x)$ を x に関する何らかの命題とするとき，「$A(0)$」と「$A(x)$ ならば $A(x+1)$」の 2 つを導けたならば，「すべての自然数 x に対して $A(x)$」を導いていよい，というものである．

数学的帰納法は，一般に「帰納法」と呼ばれる推論法則の一例であり，帰納法は何らかの集合 (あるいは述語) の帰納的定義に付随して発生するものである．自然数に対する帰納法は数学的帰納法であるが，そのほかにも，リストに対する帰納法や，二分木に対応する帰納法が考えられる．

このような論理の世界における帰納法に対応する計算の世界の登場人物を考えよう．まず，自然数など，集合 (あるいは述語) の帰納的な定義に対応するものを考えると帰納型 (inductive data) という考えに到達する．帰納型は，型を帰納的に定義したものであり，たとえば，

$$\frac{}{\Gamma \vdash 0 : Nat} \qquad \frac{\Gamma \vdash x : Nat}{\Gamma \vdash s(x) : Nat}$$

という 2 つの規則で生成されるのが，自然数をあらわす Nat 型であり，

$$\frac{}{\Gamma \vdash <> : NatList} \qquad \frac{\Gamma \vdash n : Nat \quad \Gamma \vdash L : NatList}{\Gamma \vdash < n, L > : Nat}$$

という 2 つの規則で生成されるのが，自然数のリストをあらわす NatList 型である．このように，帰納的に型 (データ型) を定義することにより，型の表現力が非常に強くなることに注意されたい．

さて，帰納型が定義できたので，次に帰納法に対応するルールを考えよう．自然数の場合，

$$\frac{\Gamma \vdash n : Nat \quad \Gamma \vdash f : Nat \to T \to T \quad \Gamma \vdash a : T}{\Gamma \vdash Rec(n, f, a) : T}$$

という形になる．やや複雑な形に見えるかもしれないが，これは帰納法のルールを非常に自然に表現したものである．数学的帰納法にあらわれる $A(x)$ という命題がここでは T という型で表現され，(A が任意の述語でよかったように T も任意の型でよい)，$a : T$ は $A(0)$ の証拠に相当し，$f : Nat \to T \to T$ は $\forall n : Nat(A(n) \supset A(n+1))$ の証拠に相当する．すなわち，f は，自然数と $A(n)$ の証拠が与えられると $A(n+1)$ の証拠を返す関数である．

上記の Rec という関数はこのルールのために新たに導入された記号であり，以下の計算規則をもつ．(この計算規則は勝手に決めたものではなく，帰納型の定義から自動的に生成されるものであるが，ここでは詳細は省略する．)

$$Rec(0, f, a) \to a$$

$$Rec(s(n), f, a) \to f(n, Rec(n, f, a))$$

すなわち $R(n, f, a)$ の計算は，n により場合分けされ，n が 0 のときは a そのものを返し (a は $A(0)$ の証拠であったことに注意せよ)，n が 0 より大きいときは，f を n 回繰返し適用して $A(n)$ の証拠を得るものである．

上と同様に，リスト型に対する Rec 関数や二分木型に対する Rec 関数を定義することができる．このように，論理における帰納は，計算における帰納型 (および Rec 関数による計算) に対応するといえる．

補足: より強力な繰返し機構について

本節で紹介したのは，繰返し計算機構の中でも最も単純な原始帰納法に対応するものである．原始帰納法は，上記の Rec 関数を見てわかるように $n = 0, 1, 2, \cdots$ に対する値を順番に計算していくので，単純な for ループに対応していることがわかる．一般の再帰呼び出しはたとえば，

$$f(x) \ = \ if \ x = 1 \ then \ 1 \ else \ if \ even(x) \ then \ f(x/2) \ else \ f(3x + 1)$$

のように，x における f の値が x より前の値とは限らない y に対する値 $f(y)$ に依存している．このような再帰を一般再帰と呼ぶ．原始帰納による計算が必ず停止するのに対して，一般再帰は，計算が必ずしも停止しない．論理に対応する計算体系は，計算が停止するもののみを考えるため，一般再帰に対応する論理は存在しない．

9.4.3 その 3: 2 階論理と多相型

多相型 (polymorphic type) とは，型の世界を拡張したものである．まず例を考える．

型なしラムダ計算の例で出てきた，$double = \lambda f.\lambda x.f(f(x))$ というラムダ式は，「関数 f をもらって，その効果を 2 倍にする関数を返すような高階の関数」を表していた．これを型付きラムダ計算の世界で考えると，

$$\vdash \lambda f : A \to A.\lambda x : A.f(f(x)) \quad : \quad (A \to A) \to (A \to A)$$

という型付けを持つことがわかる．ところで，ここで A は何であろうか？明らかにどのような型を A のところに代入しても，上記のラムダ式はその型を持つ．つまり，上のラムダ式は実質的に「任意の型 A に対して $(A \to A) \to (A \to A)$ という型をもつ」はずである．

ところが，この講義で扱った範囲では，このような「任意の型」という表現はなかったので，このことは表現できない．したがって，この講義で扱った体系をプログラム言語と考えると，「上記の A を自然数の型にした場合」や「上記の A を文字列の型にした場合」などを別々に定義しなければならず，実質的に同じプログラムであるはずなのに，何回もプログラムを書かなければならない，という不具合が生じる．

そこで，「任意の型」を表現できるように，型の世界を拡張することを考える．ちなみに，ここで言う「任意の型」は，変化するものが型であるので，前々節で与えたような Π 型では表現できない．(Π 型は，変化するものが型ではなく，自然数など「項で表現されるもの」であった)．しかし，「任意」であることにはかわらないので，$\forall X$ と書いてしまおう．すなわち，$\forall X.((X \to X) \to (X \to X))$ というような型を許すのである．この型の意味は「どんな型 X に対しても $((X \to X) \to (X \to X))$ という型を持つ項の型」というものである．これを使うと上記のラムダ式は，

$$\vdash \lambda f : X \to X.\lambda x : X.f(f(x)) \quad : \quad \forall X.((X \to X) \to (X \to X))$$

と型付けできることになる．これによって，この 1 つのラムダ式を使って，「X を自然数の型にした場合」や「X を文字列の型にした場合」などが使える．

このような型を多相型 (polymorphic type) と呼び，現代的なプログラム言語では非常に重要な概念となっている．(ML 言語では，制限された多相型が導入されている．)

多相型も Curry-Howard の同型対応によって論理の世界に対応付けることができる．「型」は「命題」に対応付けられたので，「全ての型」は「全ての命題」に対応付けられる．すなわち，多相型に対応する論理体系は，「全ての命題について...」という表現を持つ強力な体系である．(一階述語論理では「全ての自然数につ

いて...」という表現はできたが，「全ての自然数上の命題について...」という表現はできなかったことに気付いてほしい．)

このように，「命題」や「述語」を表す変数を持ち，その変数を限量子によって束縛することのできる論理を二階論理という．(二階論理における命題をさらに束縛すれば 3 階論理になる等の階層があるが，ここでは立ちいらない．) 二階論理は，一階論理に比べて，本当に表現力が強くなる (圧倒的に強くなる) ことが知られている．二階論理はそれ自体非常に興味深い．なぜなら，人間は知らず知らずのうちに一階論理の範囲を越えた推論を平気でやっているからである．また，プログラム言語の世界での多相型は，型システムの重要性の高まりとともに非常に重要な概念となりつつある．

9.4.4 その 4: 古典論理とコントロールオペレータ

本講義で扱った内容は，すべて直観主義論理であった．すなわち，計算 (コンピュテーション) の論理ではあったが，排中律 ($A \vee \neg A$) や 二重否定除去の規則 ($(\neg\neg A) \supset A$) が成立しない，という意味で人間の通常の推論からは「異常」な世界であった．

では，これらの規則に (Curry-Howard 対応の意味で) 対応するような計算の論理はないのであろうか？この疑問は昔は真面目に考えられることはなかったが，1980 年頃の Griffin という研究者の研究をきっかけに急速に研究が深まり，今日では極めて豊富な結果が得られている．

その中身は，ここで述べるには高度になり過ぎるので，雰囲気だけを箇条書きする．

- 排中律や二重否定除去の規則に対応するのは，普通のラムダ計算の世界の中のものではなく，コントロールオペレータと呼ばれるものである．
- コントロールオペレータとは，ラムダ計算などの関数型プログラム言語における制御演算子である．(制御演算子とは，C 言語や FORTRAN 言語においては，GOTO や IF-THEN-ELSE や WHILE など，実行の順序を変更する構文のことである．)
- コントロールオペレータには，例外 (JAVA 言語, ML 言語など), キャッチスロー (Common Lisp 言語など), 継続 (Scheme, Standard ML 言語など) などがある．

実行の順序を変更する機構があると，なぜ，古典論理に対応するのか？簡単に説明するのは難しいが，直感的に言うと: JAVA の例外 (exception) 機構を例にとると，例外機構を使ったプログラムの実行では，「通常終了 (例外が起きずに計算が終わる)」と「例外発生」の 2 通りの終わり方がある．これを利用すると，「A が成立するときは B を行い，$\neg A$ が成立するときは C を行う」といった場合分けの論法に対応する「証拠」を計算することができる．$A \vee \neg A$ という排中律は場合分けの論法に対応しているので，例外機構を使って，排中律の証拠を書くことができれば，古典論理に対応した論理になるであろう．

10　応用とまとめ

10.1　応用

本講義で述べた「計算の論理」は，型理論と論理体系の対応付けを考えるものであった．型理論 (プログラム言語における型システムに関する理論) は，現代的プログラム言語に必須の重要な基礎理論であり，その応用は多方面にわたっている．

そのうち最も重要な応用は，型を利用したプログラムの解析，検証である．型システムの健全性定理は，「プログラムが実行前に型を整合的に持てば，実行中，実行後に常に型を整合的に持つ」ということを意味していた．従って，型システムを適切に設計して，健全性定理を証明すれば，「プログラムの実行中に，良い状態を保つ (悪い状態にならない)」ことを保証することができる．

このような考え方を応用して，ある種のプログラムの検証を行うことができる．たとえば，「型エラーを起こさない」「スタックを無限に消費しない」など基本的な性質のほか，「確保した範囲のメモリ以外にアクセスしない」「機密情報が漏れない (機密情報を勝手に読みださない)」といったセキュリティ上重要な性質も含まれる．「確保した範囲のメモリ以外にアクセスしない」という性質 (メモリの安全性) は，地味な性質に見えるかもしれないが，現代のセキュリティ破りの半分以上は buffer overflow 攻撃 (確保した範囲を越えて配列にアクセスする) というものであるので，実用上は極めて重要である．この性質を保証するためには，「拡張」で述べた「依存型」が活躍する (「長さのわからない配列」ではなく「長さ n の配列」を表す型が表現できるので，メモリの範囲を型のレベルで計算することができる．)

このほかに，型システムの応用は広く，コンパイラの最適化に利用したり，証明からのプログラム抽出に応用することができる．

10.2　まとめ

この講義では，構文的な (機械的な) 操作に着目し，演習を通じて理解を深めてきた．また，これらの背景となる技術・理論について述べてきた．具体的な定理や結果は別として，これらの過程の基本的な考え方，立場を以下にまとめる．

- 導出ゲーム
  導出ゲームごとに，具体的な規則は異なるが (しかも，同じ目的のゲームであってもゲーム設計者ごとに異なるが)，ゲーム (導出規則) を定めれば，「その有限回の操作によって帰納的に生成されるもの」として導出が定まる，ということは共通している．
- 「導出」という形式的操作により，いろいろな概念を形式的に定めることができること
  プログラムの構文，論理における証明などは，全て，「導出」によって形式的に定めることができる; さらに，プログラムの計算，論理における証明の計算，など，もともと「意味論」に属する概念も，「導出」によって形式的に定めることができる．これにより，プログラムや論理に対する操作を計算機上に実装することが可能になる．(プログラムの型推論・計算・検証，定理の自動証明など)
- 型システムと論理体系の関係
  型付きラムダ計算の体系と (直観主義) 論理の体系は本質的に同じである．この対応関係を利用して，各種の拡張など，片方での知見を他方へ移転することができる．プログラム言語を理解することは，そ

の論理を理解することと等価である.

付録 A　Coq システムを使った演習

2013 年度までの本講義では、自前の CAL システム[*20]を使った演習を行ってきた。このシステムは現在でも利用可能ではあるが、いくぶん古くなってきたため、2014 年度から、汎用の定理証明支援システムである Coq を利用し、この上に、命題論理や型付きラムダ計算の体系を構築して演習を行なうこととした。

これらのシステムを利用する目的は、形式的体系のチェックやアルゴリズムで処理できる部分を、ソフトウェアに委ねることにより、演習効果を高めることである。特に、紙と鉛筆の演習では、学生の解答が正しいかどうかを人手で判定する必要があるが、これらのソフトウェアを使うことにより，自分の好きな時に，好きなペースで，演習を行うことができるようになる．

A.1　Coq システムに必要な環境

プログラムを書く場合に、プログラム言語 (の処理系) 以外に、プログラムを記述/編集/実行/ディバッグ等をやりやすくするための IDE と呼ばれるシステムを利用することが多い。

それと同様に、Coq システムを利用する場合にも IDE を利用するのがよい (IDE なしで Coq の証明を書くことはできるが、開発効率が相当に悪くなる)。Coq では、ProofGeneral と CoqIDE と呼ばれる 2 つの IDE があるが、ここでは、ProofGeneral を使うこととする。(ProofGeneral 自体は、“General” という名前があらわしている通り、Coq に限らず、いろいろな定理証明支援系の IDE となることのできる汎用のシステムである。)

ProofGeneral は emacs の上で動作するので、まとめると、Coq + ProofGeneral + emacs というソフトウェアの上で Coq を使うことになる。(なお、本演習の解答を、CoqIDE を用いて作成することは可能であるので、自分で CoqIDE の解説を読んでそちらを使ってもよい。)

A.2　準備

Coq+ProofGeneral システムを使うための準備は、ごく簡単であり、自分の .emacs ファイル (ホームディレクトリ直下に置かれる) に ProofGeneral を利用するための設定を追加するだけである。この記述は、たとえば、以下のようなものである。

```
(load "/opt/local/share/ProofGeneral/generic/proof-site.el")
(defadvice coq-mode-config (after deactivate-holes-mode () activate)
  "Deactivate holes-mode when coq-mode is activated."
  (progn (holes-mode 0))
)
(add-hook 'proof-mode-hook
   '(lambda ()
      (define-key proof-mode-map (kbd "C-c C-j") 'proof-goto-point)))
```

[*20] Computation and Logic, 京都大学の佐藤雅彦教授が中心となり亀山らが協力して構築してきた教育用ソフトウェア.

なお、上記の記述は、ProofGeneral が置かれた場所によって若干異なるので、演習の際の手引き (ウェブページ) を参考にされたい。

2014 年 10 月現在で、coins システムには、Coq version 8.4pl4 がインストールされている。(自分のノート PC 上に Coq 等を載せて演習をやりたい人は、同じバージョンを載せるようにしてほしい。)

A.3 命題論理の世界 (ProgLogic)

Coq の中に命題論理の世界を構築し、命題論理における証明の演習を行えるようにした。Coq 自身は命題論理を包含しているが、Coq が持っている命題論理の機能は、この授業では使わず、それとは別に命題論理の世界を構築していることに注意されたい。このため、この演習での論理記号や証明のための命令は、Coq におけるそれらとは異なっている。

論理式について、講義資料上の表現とシステムでの表現の対応は以下の通りである。

講義資料での表現	演習システムでの表現	説明
$\bot$	`False`	「矛盾」を意味する論理式
P, Q, R	`P, Q, R`	原子命題 (基本命題)
$A \wedge B$	`A /\ B`	「かつ」
$A \vee B$	`A \/ B`	「または」
$A \supset B$	`A -> B`	「ならば」
$\neg A$	`~A`	「でない」
$A, B, C \vdash B$	`[A;B;C] |- P`	判断 (judgment)

表 3　論理式の表記の対応

次に、命題論理の証明で使う推論規則の対応を述べる。

講義資料での規則	演習システムでの規則	説明
assume	assume.	仮定をする規則
$\supset I$	`impI.`	「ならば」導入規則
$\supset E$	`impE (A).`	「ならば」除去規則、$A \supset B$ の A を指定する
$\wedge I$	`andI.`	「かつ」導入規則
$\wedge EL$	`andEL (B).`	「かつ」除去 (左) 規則、$A \wedge B$ の B を指定する。
$\wedge ER$	`andER (A).`	「かつ」除去 (右) 規則、$A \wedge B$ の A を指定する。
$\vee IL$	`orIL.`	「または」導入 (左) 規則。
$\vee IR$	`orIR.`	「または」導入 (右) 規則 。
$\vee E$	`orE (A) (B).`	「または」除去規則、$A \vee B$ の A と B を指定する。。
$\neg I$	`notI.`	「でない」導入規則。
$\neg E$	`notE (A).`	「でない」除去規則、$\neg A$ の A を指定する。
$\bot E$	`falseE.`	「矛盾」除去規則
$\neg\neg E$	`notnotE.`	二重否定除去規則 (古典論理でのみ使える)

表 4　推論規則の表記の対応

なお、今回のシステムでは、日本語文字 (全角文字) は一切使っていない。たとえば、「ならば」の記号は、->という 2 文字であらわすのであって、「→」という文字は使わない。

A.4 単純型付きラムダ計算の世界 (SimpleType)

Coq の中に、単純型付きラムダ計算 (ただ、直積型と関数型を持つもの) を構築した。

型について、講義資料上の表現とシステムでの表現の対応は以下の通りである。

講義資料での表現	演習システムでの表現	説明
$\bot$	`Empty`	空の型
S, T, U	`S, T, U`	型変数 (あるいは型定数)
$A \times B$	`A * B`	直積型
$A + B$	`A + B`	直和型
$A \to B$	`A -> B`	関数型
$\neg A$	`A -> Empty`	関数と空の型で記述できる
$M : T$	`M \in T`	M は T 型
$x : A, y : B \vdash M : U$	`[x \in A; y \in B] |- M \in U`	判断 (judgment)

表 5　型の表記の対応

次に、項について、講義資料上の表現とシステムでの表現の対応は以下の通りである。

講義資料での表現	演習システムでの表現	説明
(M, N) (または $\langle M, N \rangle$)	`(M,N)`	対
$left(M)$	`left(M)`	左への射影
$right(M)$	`right(M)`	右への射影
$inl(M)$	`inl(M)`	直和への埋め込み
$inr(M)$	`inr(M)`	直和への埋め込み
$case(x, (y : B)M, (z : C)N)$	`case(x,(y \in B)M,(z \in C)N)`	場合分け
$\lambda(x : A)M$	`\lambda (x:A) M`	関数 (ラムダ式)
$M@N$ (または $M\ N$)	`M @ N`	関数の適用 (使用)
$abort(M)$	`abort(M)`	実行の中断

表 6　項の対応

最後に、型付け規則について、講義資料上の表現とシステムでの表現の対応は以下の通りである。

規則の名前にはすべて大文字の R がついていて、Rvar のようなっていることに注意せよ。これは、Case や left という Coq の命令があるため、それらの重複を避けるためである。

演習システム (Coq の中に作った SimpleType の世界) は、parser については十分こなれていないため、ラムダ式を入力したとき、予想外の括弧付けをしてしまうことがある。今回の演習では、括弧を多目につけて対処してほしい。

上記のほか、SimpleType.v の後ろの方の問題では、解くべき問題に `exists_term t,` と記載され、ラムダ式の部分が `t` という変数になっている。これは、その部分に適切なラムダ式をいれる、という部分も解答者

講義資料での表現	演習システムでの表現	説明
var	`Rvar`	変数導入
pair	`Rpair`	対
left	`Rleft (S)`	$T \times S$ の T への射影
right	`Rright (T)`	$T \times S$ の S への射影
inl	`Rinl`	直和への埋め込み (左から)
inr	`Rinr`	直和への埋め込み (右から)
case	`Rcase`	場合分け
apply	`Rapply (T)`	関数適用,$T \to S$ の T を指定
lambda	`Rlambda`	ラムダ式
abort	`Rabort`	異常終了

表 7　型付け規則の対応

が考えなければいけない。たとえば、以下のセッションは、この種の問題に対する解答である。

```
Theorem ex133 : exists_term t, [] |- t \in T -> T.
Proof.
  Ex (\lambda (x \in T) x).
  Rlambda. Rvar.
Qed.
```

証明 1 行目の Ex ではじまる部分で、t の具体形を入力している。なお、演習システムでは、ラムダ式の中の変数は、英字の小文字 1 文字に限定している。(a や x は変数になるが、a1 や x134 は変数ではない。)

A.5　関数型プログラム言語の体系 (CoreML)

関数型プログラム言語の体系 (CoreML) についても型付けの演習問題を用意した。ただし、こちらは、Coq のプリミティブなキーワードとぶつかるものがあり、「講義 (あるいは本資料) のキーワード」と「演習システムにおけるキーワード」とが、大幅にちがっているので注意してほしい。

型と判断について、講義資料上の表現とシステムでの表現の対応は以下の通りである。

講義資料での表現	演習システムでの表現	説明
`int`	`int`	整数型
`bool`	`bool`	真理値型
$A \times B$	`A * B`	直積型
$A \to B$	`A -> B`	関数型
$M : T$	`M \in T`	M は T 型
$x : A, y : B \vdash M : U$	`[x \in A; y \in B] |- M \in U`	判断 (judgment)

表 8　型と判断の表記の対応

項について、講義資料上の表現とシステムでの表現の対応は以下の通りである。

講義資料での表現	演習システムでの表現	説明
10	`(% 10)`	整数定数
-13	`(% -13)`	整数定数
`true`	`_true`	真理値定数
`false`	`_false`	真理値定数
$M = N$	`M = N`	比較 (等しさ)
$M > N$	`M > N`	比較 (大なり)
(M, N)	`(M , N)`	対
`left`(M)	`_left M`	左への射影
`right`(M)	`_right M`	右への射影
$\lambda x.M$	`_lambda (x) M`	関数 (ラムダ式)
$M@N$	`M @ N`	関数の適用 (使用)
`if` L `then` M `else` N	`_if L then M else N`	条件式
`let` $x = M$ `in` N	`_let x = M in N`	let 式
`fix` $f.x.N$	`_fix f (x) M`	再帰関数

表 9　項の対応

ほとんどのキーワードの先頭に underscore (_ という記号のこと) がついていることに注意してほしい。なお、`_lambda` は長いので `_lam` と書いてもよい。ラムダ式と再帰関数において、引数をあらわす `(x)` の括弧は省略できない。

型付け規則について、講義資料上の表現とシステムでの表現の対応は以下の通りである。

講義資料での表現	演習システムでの表現	説明
var	`Rvar`	変数
int	`Rint`	整数定数変数
bool	`Rbool`	真理値定数
plus	`Rplus`	加算
eq	`Req`	比較 (等しさ)
comp	`Rcomp`	比較 (大なり)
pair	`Rpair`	対
left	`Rleft (S)`	$T \times S$ から T への射影
right	`Rright (T)`	$T \times S$ から S への射影
lambda	`Rlambda`	ラムダ式
apply	`Rapply (T)`	関数適用、$T \to S$ の T を指定
if	`Rif`	条件式
let	`Rlet (S)`	let 式、束縛変数の型を指定
fix	`Rfix`	再帰関数

表 10　型付け規則の対応

規則の名前にはすべて大文字の R がついていて、Rvar のようなっていることに注意せよ。

演習システム (Coq の中に作った SimpleType の世界) は、parser については十分こなれていないため、ラムダ式などを入力したとき、予想外の括弧付けをしてしまうことがある。今回の演習では、括弧を多目につけて対処してほしい。

A.6　関数型プログラム言語の体系 (CoreML) における計算の定式化 (EvalDef.v, Eval.v)

この章では、関数型プログラム言語の体系 (CoreML) における計算 (Evaluation) の定式化について (特に講義資料上の表現とシステムでの表現の対応について) 述べる。

項については、前節と同様であるが、念のため、再掲しておく。

講義資料での表現	演習システムでの表現	説明
10	`(% 10)`	整数定数
-13	`(% -13)`	整数定数
`true`	`_true`	真理値定数
`false`	`_false`	真理値定数
$M = N$	`M = N`	比較 (等しさ)
$M > N$	`M > N`	比較 (大なり)
(M, N)	`(M , N)`	対
`left`(M)	`_left M`	左への射影
`right`(M)	`_right M`	右への射影
$\lambda x.M$	`_lambda (x) M`	関数 (ラムダ式)
$M@N$	`M @ N`	関数の適用 (使用)
`if` L `then` M `else` N	`_if L then M else N`	条件式
`let` $x = M$ `in` N	`_let x = M in N`	let 式
`fix` $f.x.N$	`_fix f (x) M`	再帰関数

表 11　項の対応

ほとんどのキーワードの先頭に underscore (_ という記号のこと) がついていることに注意してほしい。ラムダ式と再帰関数において、引数をあらわす `(x)` の括弧は省略できない。

計算規則の表現においては、項の形になっていない「値」や、実行時環境も必要なので、それの表現も与える。

講義資料での表現	演習システムでの表現	説明
10	`(#10)`	値としての整数定数
$true$	`(#true)`	値としての整数定数
`clos`(x, M, E)	`clos(x,M,E)`	関数クロージャ
`rclos`(f, x, M, E)	`rclos(f,x,M,E)`	再帰関数のクロージャ
$[\,]$	`initE` あるいは `[]`	空の実行時環境
$E[x \mapsto v]$	`extend(E,x,v)`	実行時環境

「項としての 10」は `%10` で、「値 (計算結果) としての 10」は `#10` であることに注意されたい。

計算規則の判断 $E \triangleright M \downarrow V$ は、演習システムでは、`E >> M --> V` の形で表現する。たとえば、`initE >> (%10 + %20) --> #30` が成立する。

計算の規則について、講義資料上の表現とシステムでの表現の対応は以下の通りである。

講義資料での表現	演習システムでの表現	説明
var	`Evar`	変数
int	`Eint`	整数定数
bool	`Ebool`	真理値定数
plus	`Eplus (v1) (v2)`	加算
comp1	`Ecomp1 (v1) (v2)`	比較 (大なり)、真になるとき
comp2	`Ecomp2 (v1) (v2)`	比較 (大なり)、偽になるとき
eq1	`Eeq1 (v1) (v2)`	比較 (等しさ)、真になるとき
eq2	`Eeq2 (v1) (v2)`	比較 (等しさ)、偽になるとき
if1	`Eif1`	条件式、真になるとき
if2	`Eif2`	条件式、偽になるとき
pair	`Epair`	対
left	`Eleft`	対の左への射影
right	`Eright`	対の右への射影
lambda	`Elambda`	ラムダ式
fix	`Efix`	再帰関数
apply1	`Eapply1 (x) (t) (E) (v)`	関数適用、関数部分がラムダ式のとき
apply2	`Eapply2 (f) (x) (t) (E) (v)`	関数適用、関数部分が再帰関数のとき
let	`Elet (v)`	let 式

表 12　型付け規則の対応

規則の名前にはすべて大文字の E がついていて、Evar のようなっていることに注意せよ。

Eplus, Ecomp1, Ecomp2, Eeq1, Eeq2 の規則の引数 v1,v2 は、それぞれ引数を計算した結果の値である。(引数ではなく、計算結果であることに注意せよ。つまり、これらの規則を適用するためには、「引数の計算の最終結果が何になるか」をあらかじめ考えなければいけない。)

Eapply1 では、計算したい項が $M@N$ のとき、M を計算した結果は、関数クロージャ $clos(x,t,E)$ の形になることが期待され、また、引数 N を計算した結果は、値 v になることが期待されるが、これらの x,t,E,v を、Eapply1 の引数としてユーザが与える必要がある。

Eapply2 も同様で、計算したい項が $M@N$ のとき、M を計算した結果は、再帰関数クロージャ $rclos(f,x,t,E)$ の形になることが期待され、引数 N を計算した結果は、値 v になることが期待されるが、これらの f,x,t,E,v を、Eapply2 の引数としてユーザが与える必要がある。

Elet では `let` $x = M$ `in` L の形の式を評価するとき，M を計算した結果 V を引数としてユーザが与える必要がある.

Elambda, Eapply1, Eapply2 は長いので，それぞれ Elam, Eapp1, Eapp2 という別名をもっている.

さらにもう 1 つ注意事項であるが、Eplus などの計算で、講義テキストでは、「$(V_1 + V_2 = V$ のとき)」などという条件を書いていた。演習システムでは、これらの条件を明示的に書くことはないが、これらの条件のチェックが残っているときに、演習システムでは、

```
add1 (#10) (#20) = Some (#30)
```

などのゴールが残ってしまうことがある。このような等式がのこっている場合は、(規則に対応するものはないが)、「auto.」 と入力して解消してほしい。なお、

```
add1 (#10) (#20) = Some (#40)
```

のように、成立しない等式が残っている場合は、auto. とやっても、条件が解消されない。このようなときは、もっと前の段階で、規則の適用を誤っていたことになる。

演習システムは、parser については十分こなれていないため、ラムダ式などを入力したとき、予想外の括弧付けをしてしまうことがある。今回の演習では、括弧を多目につけて対処してほしい。