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．