7.6 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．

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

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のとき」という条件を付けており、それを満たした場合には、簡単な代入の定義となっている。一方、型のないラムダ計算の場合には、この条件を満たさないときにも代入を定義しようとして、少し複雑な定義になっていた。同様に、case の形の項に対する定義もここでは簡単になっている。

厳密に言えば、上記の簡潔な定義をする見返りに、α 同値という概念を導入する必要があるが、これについては後の節で詳述する。

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]

$$
\begin{aligned}
(\lambda x : A.M)N &\rightarrow M\{x := N\} \\
\mathtt{left}(\langle M, N \rangle) &\rightarrow M \\
\mathtt{right}(\langle M, N \rangle) &\rightarrow N \\
\mathtt{case}(\mathtt{inl}(M), x : A.N, y : B.L) &\rightarrow N\{x := M\} \\
\mathtt{case}(\mathtt{inr}(M), x : A.N, y : B.L) &\rightarrow L\{y := M\}
\end{aligned}
$$

最初の規則は型なしラムダ計算と同じ β-規則であり，関数空間の型 $A \rightarrow B$ 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，$\langle M, N \rangle$ 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．

例 11 型付きラムダ計算における計算の例をあげる．

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

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

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

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

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．

7.7 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。

多くの教科書では α 同値に基づいた定義をしている。また、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。数学では伝統的にこのような方法を取るのである。また、前の節の代入の定義で見たように、「すべての項 M, N と変数 x に対して $M\{x := N\}$ が定義できる」ために、α 同値性を導入する必要があったことも事実である。

しかし、「同じと見なす」というのは極めて厄介な概念である。たとえば、$\lambda x : A.\lambda y : B.\langle x, y\rangle$ という項と $\lambda z : A.\lambda w : B.\langle z, w\rangle$ という項とは同じとなる。見た目に違うものを同じと思うというのはどういうことか？記号表現をあつかうコンピュータサイエンスの立場からは、違う記号表現を同じと見なす立場は必ずしも相入れない。そこで、CAL システムでは、α 同値とは別の方法をとっている。それをここで紹介する。

まず，例を使ってアイディアを述べる．

$$(\lambda x : \mathtt{Int}.\ \lambda y : \mathtt{Int}.\ x + y)(y + 1)$$

という項を計算したいとする．ここで，単純に β-簡約をして，

$$\lambda y : \mathtt{Int}.\ (y + 1) + y$$

とやってしまっては束縛された y と自由であるべき y がごっちゃになってしまいまずい，というのが，話の発端であった．そこで，束縛された変数出現は名前を一斉に別のものに変えてもよいことに着目して，

$$\lambda z : \mathtt{Int}.\ (y + 1) + z$$

という項にしよう，というのが α-同値に基づく代入の定義であった．ここでは，束縛された変数の名前をかえるのをやめて，かわりに,「何番目で束縛されているか」の情報を付加することにする．つまり，上のラムダ式は，

$$\lambda y : \mathtt{Int}.\ (y1 + 1) + y0$$

となる．同様に，

$$\lambda y : \mathtt{Int}.\ \lambda y : \mathtt{Real}.\ \langle y0, \langle y1, y2\rangle\rangle$$

等の表現もできる．ここで yn というのは，n 番目の y で束縛されていることを表す．上の例では，y を束縛するものは λ だけなので，$y0$ は一番内側で束縛されており，$y1$ は外側で束縛されており，$y2$ は束縛されていない (自由である)．当然ながら，$y3$ や $y4$ という表現はあり得ない．

ここでは数字を使ったが，自然数を導入するのはやめて，$\sharp$ という文字を使って，

$$\lambda y : \mathtt{Int}.\lambda y : \mathtt{Real}.\langle y, \langle \sharp y, \sharp\sharp y\rangle\rangle$$

のように表すことにする．

この表記の利点は以下のようになる．

- 束縛される変数名がだぶらないときは，まったく普通のラムダ式と同じになる ($\sharp$ が 1 つも出てこない)．
- α 同値を考えなくてよい．
- 人間が $\sharp$ の個数を勘定してきちんと書くのは大変だが，機械的な処理は容易．

定義 10 [♯ に基づく代入] 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.

- $x\{x := N\} \equiv N$
- $y\{x := N\} \equiv y \Downarrow x$, ただし x と y は異なる変数
- $(LM)\{x := N\} \equiv (L\{x := N\})(M\{x := N\})$
- pair, left,right,inl,inr もこれと同様
- $(\lambda y : A.M)\{x := N\} \equiv \lambda y : A.\ (M\{x \Uparrow y := N \Uparrow y\})$
- case もこれと同様

ここで $N \Uparrow y$ と $y \Downarrow x$ というのは, ♯ を増やしたり減らしたりする操作で, 直感的には, 以下の意味である.

- $N \Uparrow x$ は, N の中の $\sharp^n x$ の形の変数の自由な出現に対して ♯ を 1 つ増やす.
- $N \Downarrow x$ は, N の中の $\sharp^n x$ の形の変数の自由な出現に対して ♯ を 1 つ減らす.

たとえば, $\lambda x : A.\ x + (\sharp x) + (\sharp\sharp x) \Uparrow x$ は $\lambda x : A.\ x + (\sharp\sharp x) + (\sharp\sharp\sharp x)$ であり, $\lambda x : A.\ x + (\sharp\sharp x) \Downarrow x$ は $\lambda x : A.\ x + (\sharp x)$ である. ただし, $N \Downarrow x$ は, N の中に x 自身が自由に出現するときは, 定義されない.

これらの厳密な定義は若干面倒であるが, 以下の通りである. ただし, 以下で p, q は, ♯ がつかない変数のことである.

- $\sharp^n p \Uparrow \sharp^m p \equiv \sharp^{n+1} p$, ただし $n \geq m$ のとき
- $\sharp^n p \Uparrow \sharp^m p \equiv \sharp^n p$, ただし $n < m$ のとき
- $\sharp^n p \Uparrow \sharp^m q \equiv \sharp^n p$, ただし p と q が異なる変数のとき
- $(MN) \Uparrow y \equiv (M \Uparrow y)(N \Uparrow y)$
- $(\lambda x : A.M) \Uparrow y \equiv \lambda x : A.\ (M \Uparrow (y \Uparrow x))$
- ほかの形の項に対しても同様に定義される.
- $\sharp^n p \Downarrow \sharp^n p$ は定義されない
- $\sharp^n p \Downarrow \sharp^m p \equiv \sharp^{n-1} p$, ただし $n > m$ のとき
- $\sharp^n p \Downarrow \sharp^m p \equiv \sharp^n p$, ただし $n < m$ のとき
- $\sharp^n p \Downarrow \sharp^m q \equiv \sharp^n p$, ただし p と q が異なる変数のとき

これらの定義は, かなりややこしいので例をあげる. (型の情報は関係ないので全ての型を $*$ で表す.)

$$
\begin{aligned}
& (\lambda x:*.\lambda y:*.\lambda x:*.x(\sharp x)(\sharp\sharp x))\{x := \lambda x:*.x(\sharp x)\} \\
\equiv\ & \lambda x:*.\lambda y:*.\lambda x:*.x(\sharp x)(\sharp\sharp x))\{x \Uparrow x := \lambda x:*.x(\sharp x) \Uparrow x\} \\
\equiv\ & \lambda x:*.\lambda y:*.\lambda x:*.x(\sharp x)(\sharp\sharp x))\{\sharp x := \lambda x:*.x(\sharp x) \Uparrow (x \Uparrow x)\} \\
\equiv\ & \lambda x:*.\lambda y:*.\lambda x:*.x(\sharp x)(\sharp\sharp x))\{\sharp x := \lambda x:*.x(\sharp x) \Uparrow \sharp x\} \\
\equiv\ & \lambda x:*.\lambda y:*.\lambda x:*.x(\sharp x)(\sharp\sharp x))\{\sharp x := \lambda x:*.x(\sharp\sharp x)\} \\
\equiv\ & \lambda x:*.\lambda y:*.\lambda x:*.x(\sharp x)(\sharp\sharp x))\{\sharp x := \lambda x:*.x(\sharp\sharp x)\} \\
\equiv\ & \lambda x:*.\lambda y:*.\lambda x:*.x(\sharp x)(\sharp\sharp x))\{\sharp x \Uparrow x := \lambda x:*.x(\sharp\sharp x) \Uparrow x\} \\
\equiv\ & \lambda x:*.\lambda y:*.\lambda x:*.x(\sharp x)(\sharp\sharp x))\{\sharp\sharp x := \lambda x:*.x(\sharp\sharp\sharp x)\} \\
\equiv\ & \lambda x:*.\lambda y:*.\lambda x:*.\ x(\sharp x)(\sharp\sharp x))\{\sharp\sharp x := \lambda x:*.x(\sharp\sharp\sharp x)\} \\
\equiv\ & \lambda x:*.\lambda y:*.\lambda x:*.\ x(\sharp x)(\lambda x:*.x(\sharp\sharp\sharp x))
\end{aligned}
$$

この代入の最初の項は，実質的に，

$$(\lambda w:*.\lambda y:*.\lambda z:*.zwx)\{x := \lambda u:*.ux\}$$

を表していて，最後の項は，

$$\lambda w:*.\lambda y:*.\lambda z:*.\ zw(\lambda u:*.ux)$$

を表しているので，たしかに，代入になっていることがわかる．

このような $\sharp$ の演算は，計算や論理の本質的事項ではない．実際，変数名が衝突することさえなければ，$\sharp$ を 1 つも使わず，まったく普通の代入操作ができるのである．人間が，わざわざ同じ束縛変数を違う場所では使ったりしないのは，無意識のうちに上のような操作を避けているからである．

しかし，このような操作は，コンピュータ上で変数の束縛関係を表すために必要なことである．実際，AUTOMATH という定理証明システムの先駆けとなったシステムでは，ここでの $\sharp$ と本質的に同じものが使われている．ただし，AUTOMATH では変数名を完全に消去してしまったので非常に見づらい．また，仮に，人間が書いたプログラムが $\sharp$ を 1 つも含まなかったとしても，計算の途中で $\sharp$ が登場することがある．

このような代入の導入より，α-同値性はもはや必要ないので，これ以降は，$\equiv_\alpha$ という記法は用いない．また，$\equiv$ と書けば，文字列として同じということを意味するので，$\lambda x:A.x \equiv \lambda y:A.y$ は成立しない．

7.8 値呼び計算と名前呼び計算の定式化

前節までに，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．

多数の計算戦略がある状況では，前節で与えたように「合流性」が非常に重要な性質となる．本講義で扱っている型付きラムダ計算の体系は，合流性と (強い) 停止性が成立するので，どのような順番で計算を実行しても有限時間内に必ず同一の答え (値) に到達するので，戦略を気にする必要は (あまり) ない．

一方，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，$\langle x = x+1, x = x*2\rangle$ 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。

7.9 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[10]．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，変数，$\lambda x : A.M$ の形，$\mathtt{inl}(M)$ の形，$\mathtt{inr}(M)$ の形，$\langle M, N\rangle$ 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．

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

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

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

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

$$\frac{}{\lambda x : A.M \downarrow \lambda x : A.M}\ \lambda \qquad \frac{M \downarrow \lambda x : A.L \quad N \downarrow V \quad L\{x := V\} \downarrow W}{MN \downarrow W}\ apply$$

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．

具体的な計算例については，CAL システム上の演習を参考にしてほしい．

[10]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．

7.10 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．

$$\frac{M \downarrow \lambda x : A.L \quad L\{x := N\} \downarrow W}{MN \downarrow W}\ apply$$

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．
具体的な計算例については，CAL システム上の演習を参考にしてほしい．

7.11 CBV, CBN ゲームにおけるルールの対応表

CBV ゲーム (値呼び方式における計算の導出), CBN ゲーム (名前呼び方式における計算の導出) における解答の形式は以下の通りである．

```
A[問題番号][
仮定列 ⊢ 項↓項 in CBV または CBN since
   導出
]
```

仮定列 (あるいは宣言列) は、変数 (x や A) の列である．ただし、空列でもよい．項の導出のときには X::x:A の形の仮定があったが，計算の導出の際にはそういう仮定は使わない．
CBV ゲームのルールを CAL システムにおいて表現したものは以下の通りである．

```
x ↓ x by var {}

λ (x:A)M ↓ λ (x:A)M by λ {}

M(N) ↓ V by apply {
  M ↓λ (x:A)M' の導出 ;
  N ↓ N' の導出 ;
  M'{x:=N'}↓ V の導出
}



<M,N> ↓ <M',N'> by pair {
   M ↓ M' の導出 ;
   N ↓ N' の導出
}

left(M) ↓ L by left {
   M ↓ <L,N> の導出
}

right(M) ↓ N by right {
   M ↓ <L,N> の導出
}
```

```
inl(M) ↓ inl(M') by inl {
   M ↓ M' の導出
}

inr(M) ↓ inr(M') by inr {
   M ↓ M' の導出
}

case(M,(x:A)N,(y:B)L) ↓ V by case1 {
   M ↓ inl(M') の導出 ;
   N{x:=M'} ↓ V の導出
}

case(M,(x:A)N,(y:B)L) ↓ V by case2 {
   M ↓ inr(M') の導出 ;
   L{y:=M'} ↓ V の導出
}
```

以上のルールの組み合わせによって、項の型を導出するゲームが CBV である.

CBN ゲームのルールは，CBV ゲームのルールのうち apply ルールを以下のものに変更したものである．apply ルール以外はすべて CBV ゲームと同じである.

```
M(N) ↓ V by apply {
  M ↓ λ (x:A)M' の導出 ;
  M'{x:=N}↓ V の導出
}
```

導出の前提が 3 つではなく，2 つになっていることに注意してほしい.