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。)