プログラミング言語の意味論

勝股 審也

1 意味論について

プログラミング言語の意味論とは、広い意味では文字列であるプログラムに何らかの「意味」を対応させる事である。普段、私たちはプログラムに対しコンパイラやインタプリタによって機械語や計算機の振舞いを対応づけ、それらを実際の計算機で動かすことで実用的な恩恵を受けている。

一方、プログラミング言語の理論的な研究においては、プログラムに対してその挙動を表現するような数学的な対象や構造を対応させ、それらを数学的な道具や知識を援用して分析することで、プログラムや言語の性質を調べるということをする。この対応づけを研究する分野を（狭い意味での）プログラミング言語の意味論と言い、これが私の講義のトピックである。

本講義ではプログラミング言語 PCF (Programming language for Computable Functions) の意味論を論じる。PCF は自然数とその上の高階関数の計算に特化したプログラミング言語である。意味論には様々なスタイルがあるが、今回は二通りの意味論を紹介する。一つは操作的意味論と呼ばれ、計算機の上でインタプリタを実装するのに似た自然な意味論である。もう一つは表示的意味論と呼ばれ、プログラムに対しその入出力関係を表現する関数を対応させる意味論である。この二つの意味論の間には「観測可能な範囲においてプログラムに同じ意味を与える」という、 adequacy と呼ばれる関係が成り立つ。本公開講座はこの adequacy を目指して講義を進めたい。

2 表記法

自然数の集合を $\mathbf{N}$ で表わす。集合 X から Y への写像全体の集合を $X \to Y$ で表わす。括弧が多くなると判断した場合、$f(x)$ の括弧を省略して $f\ x$ と書くことにする。この省略が続いた場合は左から順に省略が行なわれているものとする。例えば $f\ x\ y\ z$ は $((f\ x)\ y)\ z = ((f(x))(y))(z)$ となる。集合 X の元に対して集合 Y の元を一意に割り当てる対応 $x \mapsto e$ があった時、この対応が定める X から Y への写像を $\lambda x\,.\,e$ と書く。例えば対応 $x \mapsto x^2 + x + 1$ が定める写像を $\lambda x\,.\,x^2 + x + 1$ と書く。

3 プログラミング言語 PCF

3.1 PCF とは

この節では PCF を非形式的に説明する。PCF は型付きラムダ計算を基盤とした単純な文法を持つプログラミング言語である。本講義で紹介するのは**値呼び** *(call-by-value)* の PCF である。これとは別に**名前呼び** *(call-by-name)* の PCF と言うのもある。これらの区別は後程説明する。以降、M, N, L は PCF のプログラムを表わすとする。

PCF は自然数とその上の高階関数の計算に特化しており、自然数の加減と (再帰) 関数の生成や呼び出しといった機構を備えている。一方で現実のプログラミング言語が備える実用的な機能、例えばファイル入出力、画面表示、ネットワークといった機器の操作や、オブジェクトやポインタ、配列といった機構は備えていない。

PCF のプログラムの実行が停止したならば結果が必ず一つ返ってくる。PCF には以下に述べるプログラム形式がある。

1. 最も単純な PCF のプログラム形式は自然数

$$\overline{0}, \overline{1}, \overline{2}, \cdots$$

である (数学的な自然数と PCF の自然数を区別するため上線が引かれている)。自然数を実行するとそのものが結果として返ってくる。また、PCF には

$$\mathsf{succ}\ M, \qquad \mathsf{pred}\ M$$

というプログラムの形式がある。これらを実行すると M の実行結果に 1 を足した / 引いた結果が返ってくる。なお $\mathsf{pred}\ \overline{0}$ の実行結果は $\overline{0}$ である。例えば $\mathsf{succ}\ (\mathsf{succ}\ \overline{5})$ というプログラムを実行すると $\overline{7}$ が返ってくる。

実用的なプログラミング言語は自然数上の演算を豊富に持つが、PCF は succ と pred の 2 つしか持たない。しかしながら (効率の問題を除けば) これらの演算と後に述べる再帰関数形式を組み合わせることで加減乗除、もっと強く言って自然数上の計算可能な部分関数全てを表現できる。

2. 次に単純なプログラム形式は変数

$$x, y, z, \cdots$$

である。プログラムの実行時、各変数には何らかの値が割り当てられており、変数を実行するとそれに割り当てられた値が結果として返ってくる。この割り当てはプログラムの実行の進展によって動的に変わる。

例えば、x に $\overline{5}$ が割り当てられている時に $\mathsf{pred}\ x$ を実行すると $\overline{4}$ が結果として返ってくる。

3. PCF はあるプログラムの実行結果が $\overline{0}$ かどうかで条件分岐をする形式

$$\mathsf{ifzero}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N$$

を備えている。この形式を実行するとプログラム L が実行され、その結果が $\overline{0}$ かそうでないかにより M か N の実行結果が返ってくる。例えば $\mathsf{ifzero}\ (\mathsf{pred}\ x)\ \mathsf{then}\ M\ \mathsf{else}\ N$ を実行すると、x に $\overline{0}$ または $\overline{1}$ が割り当たっている場合は M の実行結果が、そうでない場合は N の実行結果が返ってくる。

4. PCF ではプログラムも計算結果となり得る。以下の関数形式

$$\mathsf{fun}(x : \tau)\ .\ M$$

を実行すると、「ある値を渡されたら、それを x に割り当てて M を実行し、その結果を返すプログラム」**そのもの**が結果として即座に返ってくる[1]。このプログラムは「$x \mapsto M$ という

[1] この時 M は実行されない。

関数」として振る舞う。PCF では関数を値として返したり、関数を値として受け取る事ができる。この機能を**高階関数**と言う。

関数形式を実行して得られる関数は、その関数形式の実行時点での変数への値の割り当てを参照する。例えば、y に $\overline{7}$ が割り当てられている時に以下の関数形式を実行すると

$$\mathsf{fun}(x : \tau)\ .\ \mathsf{succ}\ y$$

y に 7 が割り当たっている状態で x から $\mathsf{succ}\ y$ を計算する関数が返る (よって常に $\overline{8}$ を返す定数関数となる)。この関数を将来 y に異なる値が割り当てられている状況で用いたとしても、その結果は常に $\overline{8}$ となる。この仕組みを**静的束縛**と呼ぶ。

形式中の τ は**型**と呼ばれ、x に渡される値の種類を表現する。型については後で定義を述べるので、今の段階では τ は自然数を表す型 nat であるとして話を進める。

5. 関数を呼び出すには次の**適用形式**を用いる。

$$(M\ N)$$

なお、表記上の約束として、$M_1\ M_2 \cdots\ M_n$ は $((\cdots(M_1\ M_2)\cdots)\ M_n)$ のこととする。この形式を実行すると、まず N が実行され、次に M が実行され、M の実行結果が関数であるならば、それに N の実行結果を渡して呼び出し、その結果が適用形式の結果となる。

例えば、プログラム $(\mathsf{fun}(x : \mathsf{nat})\ .\ \mathsf{pred}\ x)\ (\mathsf{succ}\ \overline{3})$ (仮に L と呼ぶ) の実行を追いかけると以下の様になる。

(a) $\mathsf{succ}\ \overline{3}$ が実行される。結果は $\overline{4}$ である。

(b) $\mathsf{fun}(x : \mathsf{nat})\ .\ \mathsf{pred}\ x$ が実行される。この結果は「値 x が渡されたら $\mathsf{pred}\ x$ を計算する関数」である。これを V と名づける。

(c) V に $\overline{4}$ が渡される。これにより $\overline{4}$ が x に割り当てられて $\mathsf{pred}\ x$ が実行される。その結果 $\overline{3}$ が L の実行結果となる。

6. 最後の PCF のプログラムの形式は**再帰関数形式**

$$\mathsf{rec}\ f(x : \tau) : \tau'\ .\ M$$

である。この形式は関数形式 $\mathsf{fun}(x : \tau)\ .\ M$ と同様、実行すると「ある値を渡されたら、それを x に割り当てて M を実行し、その結果を返すプログラム」が結果として返る。このプログラムは $x \mapsto M$ という関数として振る舞うが、関数形式と異るのは M 中で自分自身を f と言う名前で用いることができる点である。例えば以下のプログラムを Dbl とする。

$$\mathsf{rec}\ dbl(x : \mathsf{nat}) : \mathsf{nat}\ .\ \mathsf{ifzero}\ x\ \mathsf{then}\ \overline{0}\ \mathsf{else}\ \mathsf{succ}\ (\mathsf{succ}\ (dbl\ (\mathsf{pred}\ x)))$$

このプログラムに $\overline{n}$ を渡して実行すると以下の様になる。

- n が 0 ならば $\overline{0}$ を返す。
- n が 0 でなければ (dbl という名前で参照される) 自分自身 Dbl に $\overline{n-1}$ を渡して呼び出し、その結果に 2 を足す。

よって、自然数 $\overline{n}$ に対して $Dbl\,\overline{n}$ の実行結果は $\overline{2n}$ となる。

適用形式 $M\ N$ を実行すると M が返す関数に N の**実行結果** (=**値**) が渡される。このように、関数を呼び出す際に引数のプログラムの実行結果を渡す方法を**値呼び** *(call-by-value)* と言う。一方、関数を呼び出す際に引数のプログラム**そのもの**を渡す方法もあり、これを**名前呼び** *(call-by-name)* と言う。多くのプログラミング言語は値呼びを採用しているが、名前呼びを採用している言語も存在し (Algol, Haskell 等)、興味深いプログラミングが可能となる。本講義では単に PCF と言った場合値呼びの PCF を指すことにする。

プログラム形式	実行結果
$\overline{0}, \overline{1}, \overline{2}, \cdots$	$\overline{0}, \overline{1}, \overline{2}, \cdots$
succ M, pred M	M の結果に 1 を足した/引いた数
x	x に割り当てられているもの
ifzero L then M else N	L の結果が $\overline{0}$ なら M の結果、そうでなければ N の結果
fun$(x : \tau)$. M	値が渡されたら、それを x に割り当てて M を実行、その結果を返す関数
$M\ N$	M の結果返る関数に N の結果を渡し実行、その結果
rec $f(x : \tau) : \tau'$. M	関数形式と同じ内容のプログラム。ただし そのプログラム自身を M の中で f という名前で呼べる

図 1: PCF のプログラムの形式のまとめ

PCF の関数形式は引数を一つしか取れない。しかし、関数形式を繰り返し用いることで多変数関数を表現することができる。例えば自然数上の二引数関数は以下のプログラムで表現できる。

$$F = \text{fun}(x : \text{nat}) \text{ . fun}(y : \text{nat}) \text{ . } M$$

このプログラムに二つの自然数 $\overline{m}, \overline{n}$ を渡すには $F\ \overline{m}\ \overline{n}$ とすれば良い。この様に、多変数関数を 1 変数関数の繰り返しで表現することを**カリー化**と言う。

PCF のプログラムの例を見てみよう。再帰関数形式を用いると関数の繰り返し呼び出しを行うことができる。例えば以下は加減乗算を行うプログラムである。

$$\begin{aligned}
\text{足し算 } add &= \text{fun}(x : \text{nat}) \text{ . rec } l(y : \text{nat}) : \text{nat} \text{ .} \\
&\quad \text{ifzero } y \text{ then } x \text{ else succ } (l \text{ (pred } y)) \\
\text{引き算 } sub &= \text{fun}(x : \text{nat}) \text{ . rec } l(y : \text{nat}) : \text{nat} \text{ .} \\
&\quad \text{ifzero } y \text{ then } x \text{ else pred } (l \text{ (pred } y)) \\
\text{掛け算 } mul &= \text{fun}(x : \text{nat}) \text{ . rec } l(y : \text{nat}) : \text{nat} \text{ .} \\
&\quad \text{ifzero } y \text{ then } \overline{0} \text{ else } add\ x \text{ } (l \text{ (pred } y))
\end{aligned}$$

PCF では加減乗算だけでなく、帰納的関数 (詳細は省略するが、チューリング機械で計算できる自然数上の部分関数全体と同値な概念である) を全て表現することができる。

3.2 PCF のプログラムに関して

プログラム L のある位置の変数 x が以下のいずれかの形式 (これらは当然 L 中に現われる) の M の部分に含まれる時、x は L 中のその位置で**束縛**されていると言う。

$$\text{fun}(x : \tau) \text{ . } M \qquad \text{rec } f(x : \tau) : \tau' \text{ . } M \qquad \text{rec } x(y : \tau) : \tau' \text{ . } M$$

一方、L のある位置の x が束縛されていない時、x は L のその位置で**自由**であると言う。

関数形式 $\mathsf{fun}(x:\tau)\,.\,M$ に対して「この関数形式の束縛変数は x である」と言う。同様に再帰関数形式 $\mathsf{rec}\ f(x:\tau):\tau'\,.\,M$ に対して「束縛変数は f と x である」と言う。

プログラム M に対して $FV(M)$ と書いて M 中のある位置で自由となる変数の集合を表わす。$FV(M)=\emptyset$ の時、M は**閉じている**と言う。逆に $FV(M)\neq\emptyset$ の時、M は**開いている**と言う。

以下の二つのプログラムは字面こそ違うものの、意味は等しいと考えられる。

$$\mathsf{fun}(x:\mathsf{nat})\,.\,\mathsf{succ}\ (\mathsf{succ}\ x) \qquad \mathsf{fun}(y:\mathsf{nat})\,.\,\mathsf{succ}\ (\mathsf{succ}\ y)$$

というのも、関数の引数名は関数の振る舞いと関係ないからである。よって我々は上の二つのプログラムを同一視する関係である α **同値関係**を導入する。まず、プログラム M 中で変数 x が自由であるような位置にある x を全て変数 z で置き換えて得られるプログラムを $M[z/x]$ で表わす事にする。例えば $((\mathsf{fun}(x:\mathsf{nat})\,.\,x)\ x)[z/x]=(\mathsf{fun}(x:\mathsf{nat})\,.\,x)\ z$ となる。次に以下のプログラム間の二項関係を考える。

$$\begin{aligned}
\mathsf{fun}(x:\tau)\,.\,M &\to_\alpha \mathsf{fun}(z:\tau)\,.\,M[z/x] \quad (z\notin FV(M))\\
\mathsf{rec}\ f(x:\tau):\tau'\,.\,M &\to_\alpha \mathsf{rec}\ f(z:\tau):\tau'\,.\,M[z/x] \quad (z\notin FV(M)\cup\{f\})\\
\mathsf{rec}\ x(y:\tau):\tau'\,.\,M &\to_\alpha \mathsf{rec}\ z(y:\tau):\tau'\,.\,M[z/x] \quad (z\notin FV(M)\cup\{y\})
\end{aligned}$$

そして α **同値関係** $=_\alpha$ を $\to_\alpha$ を含む最小の合同な[2]同値関係で定義する。以降では $M=_\alpha N$ となるプログラム M,N を同一視する。

以下の条件を *Barendregt variable convention* (BVC) と呼ぶ。

> ある文章中でプログラム $M_1,\cdots,M_n$ が言及されているとする。この時ある $M_i\ (1\le i\le n)$ 中のある位置で x が束縛されていたならば、x は $M_1,\cdots,M_n$ 中の他の位置で自由となることはない。また $M_1,\cdots,M_n$ 中の異なる位置にある (再帰) 関数形式が束縛する変数は互いに異なる。

どのようなプログラムの組 $M_1,\cdots,M_n$ に対しても BVC を満たすプログラムの組 $M'_1,\cdots,M'_n$ で $M_i=_\alpha M'_i\ (1\le i\le n)$ となるものを取る事ができるため、プログラムは常に BVC を満たすと仮定しても一般性を失わない。

3.3 PCF の型システム

PCF のプログラム形式を無造作に組み合わせると無意味なものを作ってしまう場合がある。例えば $\mathsf{succ}\ (\mathsf{fun}(x:\mathsf{nat})\,.\,x)$ の様に関数 $\mathsf{fun}(x:\mathsf{nat})\,.\,x$ に 1 を足すことには意味が無い。このようなやりとりされる情報の形が合わないプログラムの組み合わせを排除するため**型システム**を導入する。

型システムは**型付け規則**によりプログラムに**型**を割り振る。型はプログラムの実行結果の種類を表現する記号であり、以下の二つの形式がある。

1. 値が自然数であることを表す形式 nat。
2. 値が型 τ の値を受け取り型 τ' の値を返す関数であることを表す形式 $(\tau\Rightarrow\tau')$。以降 $\tau_1\Rightarrow\cdots\Rightarrow\tau_{n-1}\Rightarrow\tau_n$ と書いた場合 $(\tau_1\Rightarrow(\cdots\Rightarrow(\tau_{n-1}\Rightarrow\tau_n)\cdots))$ のこととする。

[2]プログラム間の二項関係 R が合同であるとは任意のプログラム $(M,N)\in R$ と穴の一つ開いたプログラム $C[-]$ に対して $(C[M],C[N])\in R$ が成立することである。ここで $C[M],C[N]$ はそれぞれ $C[-]$ の穴を M,N で埋めて得られるプログラムを表わす。

PCF では関数そのものを値としてやりとりできるため、型は $\mathsf{nat} \Rightarrow (\mathsf{nat} \Rightarrow \mathsf{nat})$ や $(\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{nat}$ といくらでも複雑になり得る。

型はプログラムの構造に関して再帰的に割り振られる。その割り振り方を規定するのが型付け規則である。一般に開いたプログラムに型を割り振るには自由変数に割り当てられた値の型を知らないと行なうことができない。なぜなら、変数名だけでは、それに割り当てられている値の型が分からないからである。よって、そのような情報を表現するものとして**型文脈**を導入する。型文脈は変数と型の組 $x : \tau$ の有限列で列中の変数が互いに異なるものである。型文脈を表すのにギリシャ文字 Γ を用いる事にする。$x : \tau \in \Gamma$ と書いた時、$x : \tau$ という組が Γ に含まれていることを表す事にする。

それでは型付け規則を導入する。型付け規則は $\Gamma \vdash M : \tau$ という形式 (これを**型判定**と言う) を導くための規則からなる。型判定は「型文脈 Γ の元でプログラム M が型 τ を持つ」という事柄を表現している。型付け規則は以下の記法により与えられる:

$$\frac{Y_1 \quad \cdots \quad Y_n}{X}$$

これは、「Y_1 かつ $\cdots$ かつ Y_n ならば X」という規則を表現している。

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash \overline{n} : \mathsf{nat}} \qquad \frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}\ M : \mathsf{nat}} \qquad \frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathsf{pred}\ M : \mathsf{nat}}$$

$$\frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \mathsf{fun}(x : \tau)\ .\ M : \tau \Rightarrow \tau'} \qquad \frac{\Gamma \vdash M : \tau \Rightarrow \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash M\ N : \tau'}$$

$$\frac{\Gamma \vdash L : \mathsf{nat} \quad \Gamma \vdash M : \tau \quad \Gamma \vdash N : \tau}{\Gamma \vdash \mathsf{ifzero}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N : \tau}$$

$$\frac{\Gamma, f : \tau \Rightarrow \tau', x : \tau \vdash M : \tau'}{\Gamma \vdash \mathsf{rec}\ f(x : \tau) : \tau'\ .\ M : \tau \Rightarrow \tau'}$$

以上の規則のみを有限回用いて $\Gamma \vdash M : \tau$ が導かれた時、単に「$\Gamma \vdash M : \tau$ はプログラムである」と言う事にする。

上の規則の記法を積み上げていく事で型判定を導く過程を簡潔に表現できる。例えば $\vdash (\mathsf{fun}(x : \mathsf{nat})\ .\ x)\ (\mathsf{succ}\ \overline{3}) : \mathsf{nat}$ は以下の過程を経て導かれたプログラムである。

$$\frac{\dfrac{\dfrac{x : \mathsf{nat} \in x : \mathsf{nat}}{x : \mathsf{nat} \vdash x : \mathsf{nat}}}{\vdash \mathsf{fun}(x : \mathsf{nat})\ .\ x : \mathsf{nat} \Rightarrow \mathsf{nat}} \quad \dfrac{\dfrac{}{\vdash \overline{3} : \mathsf{nat}}}{\vdash \mathsf{succ}\ \overline{3} : \mathsf{nat}}}{\vdash (\mathsf{fun}(x : \mathsf{nat})\ .\ x)\ (\mathsf{succ}\ \overline{3}) : \mathsf{nat}}$$

このような導出の表現を型判定の**導出木**と呼ぶ。

プログラムに型を割り振ることで、その実行結果の値の種類を実行せずに推測することができる。この事から型システムは**静的意味論**と呼ばれることもある。

PCF の型システムにおいてプログラムは一意な型を持つ。

補題 3.1 任意の型文脈 Γ とプログラム M と型 τ, τ' に対し $\Gamma \vdash M : \tau$ と $\Gamma \vdash M : \tau'$ が導出できたならば $\tau = \tau'$ である。

補題 3.2 任意のプログラム $\Gamma \vdash M : \tau$、型 τ' と Γ 中に一切現れない変数 x に対し $\Gamma, x : \tau' \vdash M : \tau$ はプログラムとなる。

今、$x_1:\tau_1,\cdots,x_n:\tau_n \vdash M:\tau$ と $\vdash N_i:\tau_i\ (1\le i\le n)$ をプログラムとした時、M の自由な x_i を N_i で置き換えて得られたプログラムを $M[N_1/x_1,\cdots,N_n/x_n]$ で表わすことにする[3]。

命題 3.3 上の状況において $\vdash M[N_1/x_1,\cdots,N_n/x_n]:\tau$ はプログラムである。

4 PCF の操作的意味論

操作的意味論は 3.1 節で導入した PCF の非形式的意味論を数学的に明確化したものである。
まず、以下の形式の閉じたプログラムを**値**と呼ぶことにする。

- $\overline{0},\overline{1},\overline{2},\cdots$ は値。
- $\mathsf{fun}(x:\tau)\ .\ M$ は値。

また、型 τ に対して集合 C^τ を以下で定義する。

$$C^\tau = \{V \mid \vdash V:\tau\text{はプログラム} \wedge V \text{は値}\}$$

操作的意味論は「プログラム M を実行したら停止して V という値が結果として得られる」という事柄を表わす形式 $M \Downarrow V$ を以下の規則により定義する。規則の表記法は型システムのそれと同じである。

$$\frac{}{\overline{n}\Downarrow\overline{n}} \qquad \frac{M\Downarrow\overline{n}}{\mathsf{succ}\ M\Downarrow\overline{n+1}} \qquad \frac{M\Downarrow\overline{0}}{\mathsf{pred}\ M\Downarrow\overline{0}} \qquad \frac{M\Downarrow\overline{n}\quad n\ge 1}{\mathsf{pred}\ M\Downarrow\overline{n-1}}$$

$$\frac{L\Downarrow\overline{0}\quad M\Downarrow V}{\mathsf{ifzero}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N\Downarrow V} \qquad \frac{L\Downarrow\overline{k}\quad N\Downarrow V\quad k\ne 0}{\mathsf{ifzero}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N\Downarrow V}$$

$$\frac{}{\mathsf{fun}(x:\tau)\ .\ M\Downarrow\mathsf{fun}(x:\tau)\ .\ M}$$

$$\frac{M\Downarrow\mathsf{fun}(x:\tau)\ .\ M'\quad N\Downarrow W\quad M'[W/x]\Downarrow V}{M\ N\Downarrow V}$$

$$\frac{}{\mathsf{rec}\ f(x:\tau):\tau'\ .\ M\Downarrow\mathsf{fun}(x:\tau)\ .\ M[\mathsf{rec}\ f(x:\tau):\tau'\ .\ M/f]}$$

以降では $M\Downarrow V$ と書いた時、この形式が上の規則のみを有限回用いて導かれたものとする。一方、どのような値 V をもってしても $M\Downarrow V$ とならない場合、$M\Uparrow$ と書く。

ここで紹介した操作的意味論のスタイルは *big-step semantics* と呼ばれる。これに対してプログラムの実行過程の 1 ステップを書き換えによって表現する *small-step semantics* と呼ばれるスタイルもある。

以下は $\mathit{Dbl}\,\overline{2}$ の操作的意味論による実行例である。まず、常に

$$\mathit{Dbl}\Downarrow\underline{\mathsf{fun}(x:\mathit{nat})\ .\ \mathsf{ifzero}\ x\ \mathsf{then}\ \overline{0}\ \mathsf{else}\ \mathsf{succ}\ (\mathsf{succ}\ (\mathit{Dbl}\ (\mathsf{pred}\ x)))}$$

であるので、下線を引いた部分のプログラムを W と書く。また $n\in\mathbf{N}$ に対し B_n を以下で定義する。

$$B_n = \mathsf{ifzero}\ \overline{n}\ \mathsf{then}\ \overline{0}\ \mathsf{else}\ \mathsf{succ}\ (\mathsf{succ}\ (\mathit{Dbl}\ (\mathsf{pred}\ \overline{n})))$$

[3] この置き換えにより BVC を満たさないプログラムを得る場合があるが、その時は適宜 BVC を満たす α 同値なプログラムを取り、それをもって置き換えの結果とする。

簡単な計算から $B_0 \Downarrow \overline{0}$ が分かる。よって $Dbl\,\overline{2} \Downarrow \overline{4}$ を以下の様に導ける。

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Dbl \Downarrow W} \quad \dfrac{\dfrac{\dfrac{}{\overline{1} \Downarrow \overline{1}}}{\mathsf{pred}\,\overline{1} \Downarrow \overline{0}} \quad \vdots}{B_0 \Downarrow \overline{0}}}{Dbl\,(\mathsf{pred}\,\overline{1}) \Downarrow \overline{0}}}{\mathsf{succ}\,(Dbl\,(\mathsf{pred}\,\overline{1})) \Downarrow \overline{1}}}{\mathsf{succ}\,(\mathsf{succ}\,(Dbl\,(\mathsf{pred}\,\overline{1}))) \Downarrow \overline{2}} \quad \dfrac{}{\overline{1} \Downarrow \overline{1}}}{B_1 \Downarrow \overline{2}} \quad \dfrac{}{\overline{2} \Downarrow \overline{2}}}{\mathsf{pred}\,\overline{2} \Downarrow \overline{1}} \quad \dfrac{}{Dbl \Downarrow W}}{Dbl\,(\mathsf{pred}\,\overline{2}) \Downarrow \overline{2}}}{\mathsf{succ}\,(Dbl\,(\mathsf{pred}\,\overline{2})) \Downarrow \overline{3}}}{\mathsf{succ}\,(\mathsf{succ}\,(Dbl\,(\mathsf{pred}\,\overline{2}))) \Downarrow \overline{4}} \quad \dfrac{}{\overline{2} \Downarrow \overline{2}}}{B_2 \Downarrow \overline{4}} \quad \dfrac{}{\overline{2} \Downarrow \overline{2}} \quad \dfrac{}{Dbl \Downarrow W}}{Dbl\,\overline{2} \Downarrow \overline{4}}$$

命題 4.1 *1.* 任意の値 V に対し、$V \Downarrow V$ である。

2. 任意のプログラム M に対し、$M \Downarrow V$ ならば V は値である。

3. 任意のプログラム M と値 V, V' に対し、$M \Downarrow V$ かつ $M \Downarrow V'$ ならば $V = V'$ である。

4. 任意のプログラム $\vdash M : \tau$ と値 V に対し、$M \Downarrow V$ ならば $\vdash V : \tau$ である。

上で与えた操作的意味論で $M \Uparrow$ となる理由には以下の二つがある。

1. $M \Downarrow V$ となる V を見つける途中でどの規則も適用できなくなる場合。これは M の実行時に意味の無い計算を行なおうとしてエラーが起こることに相当する。前の節で見た無意味なプログラムを操作的意味論で実行するとこの状況が起こり得る。例えば $\mathsf{succ}\,(\mathsf{fun}(x : \tau)\,.\,M) \Uparrow$ である。

2. 規則を有限回用いて $M \Downarrow V$ を導くことができない場合。これは M の実行が停止しないことに相当する。例えば $\Omega = (\mathsf{rec}\,f(x : \mathsf{nat}) : \mathsf{nat}\,.\,f\,x)$ とし、$\vdash \Omega\,\overline{0} : \mathsf{nat}$ というプログラムを考えると $\Omega\,\overline{0} \Uparrow$ である。

理由 1 と理由 2 を区別するため、**実行時エラー**の概念を取り入れて操作的意味論を展開することも可能である。実行時エラーの発生を表わす特殊な値 error を導入し、各形式において意味の無い計算を行った時や部分式の実行でエラーが発生した時にエラーを返すための規則を新たに追加するのである。例えば succ に対しては次の規則を追加する。

$$\dfrac{M \Downarrow \mathsf{fun}(x : \tau)\,.\,M'}{\mathsf{succ}\,M \Downarrow \mathsf{error}} \qquad \dfrac{M \Downarrow \mathsf{error}}{\mathsf{succ}\,M \Downarrow \mathsf{error}}$$

この拡張された操作的意味論では、あるプログラム M の実行途中で意味の無い計算が発生した場合 $M \Downarrow \mathsf{error}$ となり、M の実行が停止しない場合 $M \Uparrow$ となる。

型システムは無意味な計算を行うプログラムを排除する目的で設計された。実際、型の付くプログラムをエラー処理のある操作的意味論で実行しても error が発生しない。(エラー処理の規則を完全に網羅した後で) 以下の主張が成立する。

任意のプログラム $\vdash M : \tau$ に対し、$M \Downarrow V$ ならば $V \neq \mathsf{error}$ である。

これを**型システムの健全性**と言う[4]。健全性は安全な型付きプログラミング言語をデザインする上で一つの指針となる重要な性質である

以降では常に型が割り振られたプログラムのみを考察の対象とする。よって、型システムの健全性から $M \Uparrow$ ならばどちらの操作的意味論においても実行が停止しないために M の値が求まらない (理由 2) という状況が起こっていると考えて良い。

5 PCF の表示的意味論

操作的意味論は現実の計算機上で PCF を実行するのに良く似た意味論であり、プログラムの動作ステップの実行過程を把握するのに適しているが、プログラムの実行結果や入出力関係を把握するにはあまり向かない。

対照的に、**表示的意味論**はプログラムの実行過程を抽象化し、プログラムの入出力関係を写像として捉える意味論である。表示的意味論のアイデアを大ざっぱに (不正確に) 述べると次の様になる。

1. 型 τ に対して集合 $[\![\tau]\!]$ を割り当てる。この割り当てを**型の解釈**と言う。型の解釈を型文脈に対して以下の様に自然に拡張する。

   $$[\![x_1 : \tau_1, \cdots, x_n : \tau_n]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$$

   型文脈が空列の場合、その解釈は 1 点集合 $\{*\}$ とする。

2. プログラム $\Gamma \vdash M : \tau$ に対して写像 $[\![M]\!] \in [\![\Gamma]\!] \to [\![\tau]\!]$ を割り当てる。この割り当てを**プログラムの解釈**と言う。型文脈 Γ が空列の場合、M の解釈を $[\![\tau]\!]$ の元と同一視する。

実際には単なる集合と写像を用いて PCF を解釈しようとしても以下に挙げる問題のためうまく行かない[5]。この事を見るため、試しに型の解釈を以下で定義する。

$$[\![\mathsf{nat}]\!] = \mathbf{N}, \qquad [\![\tau \Rightarrow \tau']\!] = [\![\tau]\!] \to [\![\tau']\!]$$

1. この型の解釈だと PCF のプログラムが潜在的に持つ “実行が停止しない” 状況をうまく捉えられない。例えば $\vdash M : \mathsf{nat}$ というプログラムの実行が停止しない時、M の解釈として適切な自然数を選ぶことができない。

2. この型の解釈だと再帰関数形式を解釈するのにも不都合がある。例として、プログラム $\vdash \mathsf{rec}\, f(x : \mathsf{nat}) : M\ . : \mathsf{nat} \Rightarrow \mathsf{nat}$ がどのような関数 $f_M \in \mathbf{N} \to \mathbf{N}$ で解釈されるかを考える。この f_M は任意の $x \in \mathbf{N}$ に対して

   $$f_M(x) = [\![M]\!](f_M, x) \tag{1}$$

   を満たしていると考えられる。この事をより簡潔に表現するため関数 $P_M \in (\mathbf{N} \to \mathbf{N}) \to (\mathbf{N} \to \mathbf{N})$ を以下で定義する。

   $$P_M = \lambda h\ .\ \lambda x\ .\ [\![M]\!](h, x)$$

   すると (1) は

   $$f_M = P_M(f_M)$$

[4] この逆は成立しない; なぜか?

[5] PCF から rec 形式を抜いた言語であれば上に挙げた二つの問題は無くなるためうまく行く。しかしこの言語の表現力は PCF よりも格段に落ちる。

と同値になる。よって、再帰関数形式 rec $f(x:\tau):M$. の解釈 f_M は P_M の不動点であると考えられる。しかし関数集合 $\mathbf{N}\to\mathbf{N}$ には不動点の構成や存在を議論をするのに十分な構造(順序/位相や極限の概念) が無いため f_M を与えることができない。

これらの問題 (特に 2 番目) を解決するために Scott が導入したのは最小上界を取る操作のある半順序集合である。彼はそれらを**領域**と呼んだ。そして単なる集合を用いる代わりに領域を用いて型を解釈し、プログラムを**連続写像**で解釈することで上述の問題を解決した。この節では一番シンプルな領域理論を紹介し、PCF の表示的意味論を展開する。

5.1 初歩の領域理論

定義 5.1 1. **半順序集合**とは台集合 D と D 上の二項関係 $\sqsubseteq_D$ の組 $(D,\sqsubseteq_D)$ で以下を満たすものである。

$$\begin{aligned}
&\text{(反射律)} && \forall d\in D\,.\,d\sqsubseteq_D d\\
&\text{(推移律)} && \forall d,d',d''\in D\,.\,d\sqsubseteq_D d'\wedge d'\sqsubseteq_D d''\implies d\sqsubseteq_D d''\\
&\text{(反対称律)} && \forall d,d'\in D\,.\,d\sqsubseteq_D d'\wedge d'\sqsubseteq_D d\implies d=d'
\end{aligned}$$

2. ω-**完備半順序集合** (ω-complete partial order、略して ω-*CPO*) とは半順序集合 $(D,\sqsubseteq_D)$ で任意の単調増加な可算列 $d_0\sqsubseteq_D d_1\sqsubseteq_D\cdots$ に対して最小上界が存在するものである。最小上界は列に対して一意に定まるので、これを $\bigsqcup_{n\in\mathbf{N}} d_n$ と書く。

3. ω-**完備点付き半順序集合** (ω-complete pointed partial order、略して ω-*CPPO*) とは、ω-CPO $(D,\sqsubseteq_D)$ で最小元があるもののことである。最小限は一意に定まるので、これを $\bot_D$ と書く[6]。

以降では「D を ω-CPO (ω-CPPO) とする」と宣言した時、その台集合を同じ文字 D、順序を $\sqsubseteq_D$、(最小元を $\bot_D$) で表わすことにする。

以下は ω-CPO の例である。

1. 任意の集合 X 上の元の一致の関係 $=_X$ は明らかに反射律、推移律、反対称律を満たす。よって $(X,=_X)$ は ω-CPO となる。

2. 集合 X の部分集合全体 $P(X)$ は包含関係による順序で ω-CPPO となる。最小元は空集合、また単調増加列 $d_0\subseteq d_1\subseteq\cdots$ の最小上界は $\bigcup_{i\in\mathbf{N}} d_i$ で与えられる。

3. $\{\bot\sqsubseteq\top\}$ という ω-CPPO を *Sierpinski space* と言い、$\mathbf{O}$ で表す。

4. 自然数の集合 $\mathbf{N}$ に一点 ∞ を追加した集合 $\mathbf{N}\cup\{\infty\}$ は以下の全順序により ω-CPPO となる。これを ω で表わす。

$$0\le 1\le 2\le\cdots\le\infty$$

ω-CPO は単調増加列の最小上界を取る事のできる半順序集合である。よって ω-CPO の間の写像を定義する際、順序と最小上界を取る操作に対して考慮するのが自然である。

定義 5.2 D,D' を ω-CPO とする。

[6] ω-CPO を predomain、ω-CPPO を domain と呼ぶ教科書もある。

1. 関数 $f \in D \to D'$ が**単調**であるとは f が D の順序を保つ、つまり任意の $x, y \in D$ に対し

$$x \sqsubseteq_D y \implies f(x) \sqsubseteq_{D'} f(y)$$

が成立することである。

2. 関数 $f \in D \to D'$ が**連続**であるとは f が単調かつ D 中の任意の単調増加列の最小上界を保つ、つまり $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \cdots$ という D 中の単調増加列に対し

$$f\left(\bigsqcup_{n\in\mathbf{N}} d_n\right) = \bigsqcup_{n\in\mathbf{N}} f(d_n)$$

が成立することである。D から D' への連続関数全体の集合を $[D \to D']$ と表す。

例えば任意の $k \in \mathbf{N}$ に対して以下の関数は ω から $\mathbf{O}$ への連続関数となる。

$$\delta_k(x) = \begin{cases} \top & x \geq k \\ \bot & x < k \end{cases}$$

しかし、δ_∞ は単調であるものの連続にはならない。

5.2 ω-CPO の構成

直積 D, D' を ω-CPO とする。集合 $D \times D'$ 上に以下の順序を定める。

$$(d, d') \sqsubseteq_{D\times D'} (e, e') \iff d \sqsubseteq_D e \wedge d' \sqsubseteq_{D'} e'$$

順序集合 $(D \times D', \sqsubseteq_{D\times D'})$ は ω-CPO となる。これを $D \times D'$ と表わす。

命題 5.3 *D, D', D'' を ω-CPO とせよ。*

1. 任意の連続関数 $f \in [D \to D'], g \in [D \to D'']$ に対して

$$\lambda x\,.\,(f(x), g(x)) \in [D \to D' \times D'']$$

である。

2. 以下で定義される関数 $\pi \in D \times D' \to D, \pi' \in D \times D' \to D'$ を射影関数という。

$$\pi(x, y) = x, \qquad \pi'(x, y) = y$$

射影関数は連続である。つまり

$$\pi \in [D \times D' \to D], \qquad \pi' \in [D \times D' \to D']$$

である。

連続関数空間　D, D' を ω-CPO とする。$[D \to D']$ 上に以下の順序を定める。

$$f \sqsubseteq_{[D\to D']} g \iff \forall x \in D\,.\, f(x) \sqsubseteq_{D'} g(x)$$

順序集合 $([D \to D'], \sqsubseteq_{[D\to D']})$ は ω-CPO となる。これを $[D \to D']$ と表わす。

命題 5.4　*D, D', D'' を ω-CPO とする。*

1. 連続関数 $f \in [D \times D' \to D'']$ に対して以下で定義される関数 $\lambda(f) \in D \to (D' \to D'')$ を f のカリー化と言う。
$$\lambda(f)(x) = \lambda y\,.\, f(x, y)$$
カリー化した関数は連続である。つまり
$$\lambda(f) \in [D \to [D' \to D'']]$$
である。

2. 以下で定義される関数 $\mathbf{ev} \in ((D \to D') \times D) \to D'$ を評価関数と言う。
$$\mathbf{ev}(f, x) = f(x)$$
評価関数は連続である。つまり
$$\mathbf{ev} \in [[D \to D'] \times D \to D']$$
を満たす。

簡単な事実として、D が ω-CPO で D' が ω-CPPO の場合 $[D \to D']$ は ω-CPPO となる。

持ち上げ　D を ω-CPO とする。D に含まれない元 $\bot$ を取り、集合 $D \cup \{\bot\}$ 上に以下の順序 $\sqsubseteq_{D_\bot}$ を定める。

$$x \sqsubseteq_{D_\bot} y \iff x = \bot \vee x \sqsubseteq_D y$$

順序集合 $(D \cup \{\bot\}, \sqsubseteq_{D_\bot})$ は ω-CPPO となる。これを $D_\bot$ と表わす。また、D から $D_\bot$ への包含写像 (連続である) を $\lfloor - \rfloor$ で表わす。

定義 5.5　*D, D' を ω-CPO とせよ。このとき $seq \in [D_\bot \times [D \to D'_\bot] \to D'_\bot]$ を以下で定義する。*

$$seq(\bot, f) = \bot, \qquad seq(\lfloor x \rfloor, f) = f(x)$$

実際にはこの関数は以下の **let** 記法によって用いられる。今 e を $D_\bot$ の元、e' を $x \in D$ によって一意に定まる $D'_\bot$ の元とした場合、**let** $\lfloor x \rfloor$ **be** e **in** e' と書いて $seq(e, \lambda x\,.\, e')$ を表わすとする。

5.3　最小不動点

5 節の先頭で単なる集合とそれらの間の写像には不動点を構成するのに十分な構造が無いという問題を指摘した。一方、ω-CPPO 上の連続関数に関しては常に不動点を構成することができる。

命題 5.6　*D を ω-CPPO とし、$f \in [D \to D]$ とせよ。*

1. $\bot_D \sqsubseteq f(\bot_D) \sqsubseteq f(f(\bot_D)) \sqsubseteq \cdots \sqsubseteq f^{(n)}(\bot_D) \sqsubseteq \cdots$ という単調増加列の最小上界

$$\bigsqcup_{n\in\mathbf{N}} f^{(n)}(\bot_D)$$

は f の不動点で最小のものとなる。

2. 写像 $\mathbf{fix} : f \mapsto \bigsqcup_{n\in\mathbf{N}} f^{(n)}(\bot_D)$ は $[D \to D]$ から D への連続写像である (つまり $\mathbf{fix} \in [[D \to D] \to D]$)。

5.4 PCF の表示的意味論

準備が整ったので、PCF の領域理論による表示的意味論を与える。型とプログラムの解釈を以下で与える。

型の解釈 型 τ に対して ω-CPO $[\![\tau]\!]$ を再帰的に定義する。

$$[\![\mathsf{nat}]\!] = (\mathbf{N}, =_{\mathbf{N}}), \qquad [\![\tau \Rightarrow \tau']\!] = [[\![\tau]\!] \to [\![\tau']\!]_\bot]$$

型の解釈を型文脈の解釈に自然に拡張する。

$$[\![x_1 : \tau_1, \cdots, x_n : \tau_n]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$$

型文脈が空列の時はその解釈を一点集合 $\{*\}$ とする。

プログラムの解釈 プログラム $\Gamma \vdash M : \tau$ に対して連続関数

$$[\![M]\!] \in [[\![\Gamma]\!] \to [\![\tau]\!]_\bot]$$

を型判定の導出に関して再帰的に定義する。値域が $[\![\tau]\!]$ の持ち上げになっているのは、M の実行が停止しない事を元 $\bot$ により表現できるようにするためである。Γ が空列の時は $[\![M]\!]$ を $[\![\tau]\!]_\bot$ の元と同一視する。以下では Γ 中の i 番目の位置の変数を x_i で表わす。また、ρ を $(v_1, \cdots, v_n)$ という仮引数の略記とする。

$$\begin{aligned}
[\![x_i]\!] &= \lambda(v_1, \cdots, v_n) \,.\, \lfloor v_i \rfloor \quad (n \text{ は型文脈 } \Gamma \text{ の長さ}) \\
[\![\overline{n}]\!] &= \lambda\rho \,.\, \lfloor n \rfloor \\
[\![\mathsf{fun}(x : \tau) \,.\, M]\!] &= \lambda\rho \,.\, \lfloor \lambda x \,.\, [\![M]\!](\rho, x) \rfloor \\
[\![M\ N]\!] &= \lambda\rho \,.\, \mathbf{let}\ \lfloor n \rfloor\ \mathbf{be}\ [\![N]\!]\rho\ \mathbf{in}\ \mathbf{let}\ \lfloor m \rfloor\ \mathbf{be}\ [\![M]\!]\rho\ \mathbf{in}\ \mathbf{ev}(m, n) \\
[\![\mathsf{succ}\ M]\!] &= \lambda\rho \,.\, \mathbf{let}\ \lfloor m \rfloor\ \mathbf{be}\ [\![M]\!]\rho\ \mathbf{in}\ \lfloor m + 1 \rfloor \\
[\![\mathsf{pred}\ M]\!] &= \lambda\rho \,.\, \mathbf{let}\ \lfloor m \rfloor\ \mathbf{be}\ [\![M]\!]\rho\ \mathbf{in} \begin{cases} \lfloor 0 \rfloor & m = 0 \\ \lfloor m - 1 \rfloor & m > 0 \end{cases} \\
[\![\mathsf{ifzero}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N]\!] &= \lambda\rho \,.\, \mathbf{let}\ \lfloor l \rfloor\ \mathbf{be}\ [\![L]\!]\rho\ \mathbf{in} \begin{cases} [\![M]\!]\rho & (l = 0) \\ [\![N]\!]\rho & (l \neq 0) \end{cases} \\
[\![\mathsf{rec}\ f(x : \tau) : \tau' \,.\, M]\!] &= \lambda\rho \,.\, \lfloor \mathbf{fix}(\lambda h \,.\, \lambda v \,.\, [\![M]\!](\rho, h, v)) \rfloor
\end{aligned}$$

表示的意味論の例を見てみよう。3.1 節で見た足し算のプログラム $\vdash add : \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$ を表示的意味論で解釈すると以下の元 $[\![add]\!] \in [\mathbf{N} \to [\mathbf{N} \to \mathbf{N}_\bot]_\bot]_\bot$ となる。

$$[\![add]\!] = \lfloor \lambda x \,.\, \lfloor \mathbf{fix}(h\ x) \rfloor \rfloor$$

ここで $h \in [\mathbf{N} \to [[\mathbf{N} \to \mathbf{N}_\perp] \to [\mathbf{N} \to \mathbf{N}_\perp]]]$ は以下の連続写像である。

$$h = \lambda x\,.\,\lambda f\,.\,\lambda y\,.\,\begin{cases} \lfloor x \rfloor & (y = 0) \\ \textbf{let } \lfloor z \rfloor \textbf{ be } f(y-1) \textbf{ in } \lfloor z+1 \rfloor & (y > 0) \end{cases}$$

表示的意味論において *add* が実際に自然数上の加算となっていることを確認しよう。

命題 5.7 任意の $n, m \in \mathbf{N}$ に対して $[\![add\ \overline{n}\ \overline{m}]\!] = [\![\overline{n+m}]\!] = \lfloor n+m \rfloor$ が成立する。

証明 まず、簡単な計算から以下の等式を得る。

$$[\![add\ \overline{n}\ \overline{m}]\!] = \mathbf{fix}(h\ n)(m) = h\ n\ (\mathbf{fix}(h\ n))\ m$$

右の等号は $\mathbf{fix}(h\ n)$ が $h\ n$ の不動点であることを用いている。次に $h\ n\ (\mathbf{fix}(h\ n))\ m = \lfloor n+m \rfloor$ を m に関する帰納法で証明する。

- $m = 0$ の時。$h\ n\ (\mathbf{fix}(h\ n))\ 0 = \lfloor n \rfloor$ より良い。
- $m > 0$ の時。$h\ n\ (\mathbf{fix}(h\ n))\ m = \textbf{let } \lfloor z \rfloor \textbf{ be } \mathbf{fix}(h\ n)\ (m-1) \textbf{ in } \lfloor z+1 \rfloor$ である。帰納法の仮定から $\mathbf{fix}(h\ n)\ (m-1) = \lfloor n+m-1 \rfloor$。よって $h\ n\ (\mathbf{fix}(h\ n))\ m = \lfloor n+m-1+1 \rfloor = \lfloor n+m \rfloor$ を得る。

ゆえに $[\![add\ \overline{n}\ \overline{m}]\!] = [\![\overline{n+m}]\!] = \lfloor n+m \rfloor$ が成立する。

6 Adequacy

それではこの講義のゴールである adequacy について紹介する。PCF の adequacy とは、表示的意味論と操作的意味論が「観測可能な範囲において一致する」という主張である。

まず簡単に分かるのは以下の定理である。

定理 6.1 任意のプログラム $\vdash M : \tau$ について $M \Downarrow V$ ならば $[\![M]\!] = [\![V]\!]$ である。

証明 $M \Downarrow V$ に関する帰納法で証明する。

この定理から、表示的意味論は操作的意味論の実行ステップを正しく抽象化し、プログラムの返す値を捉えていると理解できる。しかしながら、この定理の逆は一般には成立しない。実際、

$$[\![\mathsf{fun}(x : \mathsf{nat})\ .\ \overline{0}]\!] = [\![\mathsf{fun}(x : \mathsf{nat})\ .\ \mathsf{pred}\ \overline{1}]\!] = \lfloor \lambda x\,.\,\lfloor 0 \rfloor \rfloor$$

であるが操作的意味論においては

$$\mathsf{fun}(x : \mathsf{nat})\ .\ \overline{0} \not\Downarrow \mathsf{fun}(x : \mathsf{nat})\ .\ \mathsf{pred}\ \overline{1}$$

である。

この現象は特に関数型の値で起こる。というのも操作的意味論では、関数型の値はプログラムそのものであり、表示的意味論が着目している入出力関係よりも細かな情報 (=実行手順) を有しているからである。従って、我々は操作的意味論と表示的意味論を比較する際、観測する型を全ての型ではなく、nat 型のみ制限にする。実際、現実の関数型言語においてユーザーが見る事のできる値は自然数といった基本的なデータ型のものに限られており、関数型の値は見えない。この事からも観測する型の範囲を nat 型のみに制限するのは妥当なように思われる。

以上の議論に従い、我々は操作的意味論と表示的意味論を nat 型のプログラムの実行結果により比較する。まず、定理 6.1 の帰結として次が言える。

定理 6.2 任意のプログラム $\vdash M : \mathsf{nat}$ と $n \in \mathbf{N}$ に対し $M \Downarrow \overline{n}$ ならば $[\![M]\!] = \lfloor n \rfloor$ である。

次に定理 6.2 の逆を考える。

任意のプログラム $\vdash M : \mathsf{nat}$ と $n \in \mathbf{N}$ に対し $[\![M]\!] = \lfloor n \rfloor$ ならば $M \Downarrow \overline{n}$ である。

これを示すには以下を示せば十分であることに気付く。

任意のプログラム $\vdash M : \tau$ に対し $[\![M]\!] \neq \bot$ ならばある値 $V \in C^\tau$ が存在して $M \Downarrow V$ である。

というのも nat 型の値 V, W に限り、$[\![V]\!] = [\![W]\!]$ ならば $V = W$ が言えるからである。しかし上の主張を一般化し、$\Gamma \vdash M : \tau$ の型判定の導出に関する帰納法で証明しようとすると適用形式の所でつかえてしまう。操作的意味論では適用形式の実行を以下の規則で定義しており、

$$\frac{M \Downarrow \mathsf{fun}(x : \tau) \,.\, M' \quad N \Downarrow W \quad M'[W/x] \Downarrow V}{M\ N \Downarrow V}$$

帰納法の仮定は M と N に対しては用意されているが、$M'[W/x]$ には用意されていないためである。

この問題を回避するにはより洗練された方法を考える必要がある。Plotkin は [6] において adequacy relation を導入して上の主張を証明した[7]。Adequacy relation とは型 τ に関して再帰的に定義された二項関係 $R^\tau \subseteq [\![\tau]\!] \times C^\tau$ である。

$$(n, \overline{m}) \in R^{\mathsf{nat}} \iff n = m$$
$$(f, \mathsf{fun}(x : \tau) \,.\, M) \in R^{\tau \Rightarrow \tau'} \iff \forall (v, V) \in R^\tau \,.\, (f(v), M[V/x]) \in T(R^{\tau'})$$

ただし $T(R^\tau)$ は以下で定義される $[\![\tau]\!]_\bot$ と $\{M \mid \vdash M : \tau \text{はプログラム}\}$ の間の二項関係である。

$$(x, M) \in T(R^\tau) \iff \forall y \in [\![\tau]\!] \,.\, x = \lfloor y \rfloor \implies (\exists V \,.\, M \Downarrow V \wedge (y, V) \in R^\tau)$$

このように、型に関する帰納法で作られた意味論間の関係を**論理関係**と呼ぶことがある。

補題 6.3 任意の型 τ とプログラム $\vdash M : \tau$ に対して以下が成立する。

1. $(\bot, M) \in T(R^\tau)$ である。
2. $d_0 \sqsubseteq d_1 \sqsubseteq \cdots$ が $[\![\tau]\!]_\bot$ 中の単調増加列で $(d_i, M) \in T(R^\tau)$ ならば $(\bigsqcup_{i \in \mathbf{N}} d_i, M) \in T(R^\tau)$ である。

証明 型に関する帰納法で証明する。

補題 6.4 (Basic Lemma) $x_1 : \tau_1, \cdots, x_n : \tau_n \vdash M : \tau$ をプログラムとし、$(\lfloor v_i \rfloor, M_i) \in T(R^{\tau_i})\ (1 \leq i \leq n)$ とした時 $([\![M]\!](v_1, \cdots, v_n), M[M_1/x_1, \cdots, M_n/x_n]) \in T(R^\tau)$ である。

証明 プログラム $x_1 : \tau_1, \cdots, x_n : \tau_n \vdash M : \tau$ の導出に関する帰納法で証明する。

系 6.5 任意のプログラム $\vdash M : \mathsf{nat}$ と $n \in \mathbf{N}$ に対し $[\![M]\!] = \lfloor n \rfloor$ ならば $M \Downarrow \overline{n}$ である。

定理 6.6 (Adequacy) 任意のプログラム $\vdash M : \mathsf{nat}$ と $n \in \mathbf{N}$ に対し以下が成立する。

$$[\![M]\!] = \lfloor n \rfloor \iff M \Downarrow \overline{n}$$

Adequacy により、表示的意味論はプログラムの実行過程を抽象化しつつも自然数の計算を行うプログラムの停止性に関して正しくモデルしている意味論であることが分かった。このことから、あるプログラム $\vdash M : \mathsf{nat}$ の停止性を調べる事を $[\![M]\!] = \bot$ という問題に帰着させることが可能となる。

[7] 元の adequacy relation は名前呼び PCF のために定義されていた。本講義ノートではそれを値呼び PCF のために変更してある。

7 Full Abstraction

最後に *full abstraction* について言及する。プログラム $\vdash M, N : \tau$ を操作的意味論で比較する時、τ が nat でない限り、M, N の実行結果で M, N を比較することはできない事を前節で述べた。というのも関数型の値は実行手順そのものを情報として持っているため比較する事には意味が無いからである。従って我々はプログラムが返す値を比較するというアイデアを捨て、プログラムの果たす役割を比較するというアイデアを導入する。まず M をある位置に含むプログラム $\vdash C[M] : \mathsf{nat}$ を考える。このプログラムから M を抜き取ると穴が一つ空いたプログラム $C[-]$ を得る。これを**文脈**と言う。文脈の穴に N をあてはめたプログラムを $\vdash C[N] : \mathsf{nat}$ とする。この時もし

$$C[M] \Downarrow \overline{n} \implies C[N] \Downarrow \overline{n}$$

が成立したら、M の $C[-]$ 中における役割は N によっても果たすことができる、あるいは $C[M]$ 中の M を N で置き換えても違いが観測できない、と言える。そしてこの事が任意の文脈 $\vdash C[-] : \mathsf{nat}$ に対して成立したら、我々は M の PCF での役割は N によっても果たせる、つまり任意のプログラム中の M を N で置き換えても違いが観測できないと考えられる。この時我々は $M \lesssim N$ と書くことにしよう。この二項関係 $\lesssim$ は前順序 (定義 5.1 の反射律と推移律を満たす二項関係) になるため、**文脈前順序**と呼ばれる。この前順序から作られる同値関係 $M \simeq N \iff M \lesssim N \wedge N \lesssim M$ は**文脈同値**と呼ばれる。

文脈前順序は自然な概念の定式化であるものの、それを具体的なプログラムに対して示すのは一般には難しい問題である。というのも、文脈前順序の定義は「任意の文脈について」という出だしで始まっており、また操作的意味論という具体的なモデルにおいて定義されているからである。

前節で示した adequacy から以下を導くことができる。

命題 7.1 任意の型 τ とプログラム $\vdash M, N : \tau$ に対し、$[\![M]\!] \sqsubseteq [\![N]\!]$ ならば $M \lesssim N$ である。

系 7.2 任意の型 τ とプログラム $\vdash M, N : \tau$ に対し、$[\![M]\!] = [\![N]\!]$ ならば $M \simeq N$ である。

しかし、命題 7.1 の逆は、系 7.2 の逆に反例が存在するため成立しない。反例の構築にはある型 τ の解釈 $[\![\tau]\!]$ の中から、任意の $\vdash M : \tau$ に対して $[\![M]\!] \neq x$ となる元 $x \in [\![\tau]\!]_\perp$ を見つける事がポイントとなる。その典型的な例が以下の parallel or と呼ばれるものである。

定義 7.3 以下の連続関数 $por_\tau \in [[\![\tau]\!] \to \mathbf{N}_\perp] \times [[\![\tau]\!] \to \mathbf{N}_\perp] \times [\![\tau]\!] \to \mathbf{N}_\perp]$ を定義する。

$$por_\tau(f, g, x) = \begin{cases} \lfloor 0 \rfloor & f(x) = \lfloor 0 \rfloor \\ \lfloor 0 \rfloor & g(x) = \lfloor 0 \rfloor \\ \lfloor 1 \rfloor & \exists n, m > 0 \,.\, f(x) = \lfloor n \rfloor \wedge g(x) = \lfloor m \rfloor, \\ \perp & \text{otherwise} \end{cases}$$

また、$Por_\tau = \lambda f \,.\, \lfloor \lambda g \,.\, \lfloor \lambda x \,.\, por_\tau(f, g, x) \rfloor \rfloor \in [\![(\tau \Rightarrow \mathsf{nat}) \Rightarrow (\tau \Rightarrow \mathsf{nat}) \Rightarrow \tau \Rightarrow \mathsf{nat}]\!]$ と置く。

この関数は $f(x)$ (または $g(x)$) の結果が未定義でも $g(x)$ (または $f(x)$) が $\lfloor 0 \rfloor$ を返せば $\lfloor 0 \rfloor$ を返し、$f(x), g(x)$ がどちらも $\lfloor 0 \rfloor$ 以外の数を返せば $\lfloor 1 \rfloor$ を返す。このことから、*por* は内部で $f(x)$ と $g(x)$ の計算を並行に行い、どちらかが $\lfloor 0 \rfloor$、あるいはどちらも $\lfloor 0 \rfloor$ 以外の数を返すのを待つアルゴリズムであると考えられる。しかし、このような並行計算の機構は PCF には備わっていないため、上の連続写像を PCF により与えることはできない。

命題 7.4 任意の型 τ と *PCF* のプログラム $\vdash M : (\tau \Rightarrow \mathsf{nat}) \Rightarrow (\tau \Rightarrow \mathsf{nat}) \Rightarrow \tau \Rightarrow \mathsf{nat}$ に対し $[\![M]\!] = \lfloor Por_\tau \rfloor$ となるものは存在しない。

系 7.2 の逆の反例は、プログラム $\vdash M_1, M_2 : ((\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{nat}$ で、$M_1 \simeq M_2$ かつ、$[\![M_1\ x]\!](Por_{\mathsf{nat}}) \neq [\![M_2\ x]\!](Por_{\mathsf{nat}})$ となるようなものによって与えられる。

命題 7.1 の逆が成立するような表示的意味論を *fully abstract* な表示的意味論と言う。上の反例は本講義ノートで示した表示的意味論が fully abstract で無い事を意味する。fully abstract な表示的意味論の探求は *full abstraction* **問題**と呼ばれ、プログラミング意味論の研究の大きなテーマの一つとなった。PCF の full abstraction 問題は名前呼び PCF において先に良く研究され、幾つかの解決案が提案された。

- 上の反例において、*por* が書けないのは PCF の表現能力が弱いのが問題であると考え、PCF を拡張して full abstraction を達成する方法。このアプローチは [6, 2] に見られる。
- 上の反例は型の解釈に PCF で与える事のできない元が含まれているのが原因であると考え、論理関係を用いて型の解釈から *por* のような元を排除する方法 [9, 5, 7]。
- 上の反例において、プログラムが入力をどのようにアクセスするかという側面を表示的意味論が正確にモデルしていないために *por* のような元が型の解釈に紛れ込んだと考え、プログラムの入力へのアクセスを対話する二人の間のゲームと捉えて表示的意味論を再構築する方法 [4, 1]。この表示的意味論は**ゲーム意味論**と呼ばれる。1990 年代後半から盛んに研究され、PCF に限らず様々なプログラミング言語の fully abstract な表示的意味論を与えることに成功している。

また、Milner は PCF のプログラムを文脈同値で割り、極限を追加することで fully abstract な表示的意味論を作った。この意味論は文脈同値から出発して構築されたので、満足の行く解答とは見なされなかったが、fully abstract な表示的意味論を与えるための十分条件を特定するのに役立った。

値呼び PCF においても full abstraction 問題は考察され、[8, 3] といった解決案が提案されている。

謝辞

proof reading に御協力して頂いた浅田 和之氏、星野 直彦氏、Mikhov Rossen 氏、大山 知則氏、杉本 卓也氏に深く感謝いたします。彼らのコメントは原稿の改善に非常に貢献しました。また、講義に関して有益なフィードバックを下さった阿部 健氏に深く感謝いたします。

参考文献

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[2] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Inf. Comput.*, 111(2):297–401, 1994.

[3] K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation. *Theor. Comput. Sci.*, 221(1-2):393–456, 1999.

[4] J. Hyland and C.-H. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[5] P. O'Hearn and J. Riecke. Kripke logical relations and PCF. *Inf. Comput.*, 120(1):107–116, 1995.

[6] G. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5:223–257, 1977.

[7] J. Riecke and A. Sandholm. A relational account of call-by-value sequentiality. *Inf. Comput.*, 179(2):296–331, 2002.

[8] K. Sieber. Relating full abstraction results for different programming languages. In *FST and TC 10: Proceedings of the tenth conference on Foundations of software technology and theoretical computer science*, pages 373–387, New York, NY, USA, 1990. Springer-Verlag.

[9] K. Sieber. Reasoning about sequential functions via logical relations. In *Proc. LMS Symposium on Applications of Categories in Computer Science*, LMS Lecture Note Series 177, pages 258–269. Cambridge University Press, 1992.