解　説

3. 情報工学に見られる不動点論の散策

3.3 プログラム理論と不動点

——表示的意味論における不動点の役割†——

立　木　秀　樹††

1. プログラムの意味論とは

プログラムが与えられたとき，そのプログラムの意味（たとえば，これは引数の階乗を求めるプログラムだとか，これはコンパイラのプログラムだとか）は，どのようにすれば分かるだろうか．

どのようなプログラム言語でも，その言語のプログラムを実行するための計算規則が存在しており，プログラムはその計算規則を用いてなんらかのアルゴリズムを表現しているはずである．そのように考えれば，プログラムの意味は，実際に仮想的な計算機になったつもりでプログラムの実行を追っていけば得られるはずである．このように，プログラム言語の意味を計算規則により与えることを，操作的意味論という．しかし，これでは，プログラムの意味は計算機にかけてみないと分からないと言っているに等しい．

もし，実行過程とは独立した方法で，プログラムの意味を静的に解釈することができ，その意味と合致する結果を得る計算方法が別にあって，計算はそれにより行われるものならば，プログラムの意味は計算過程を追わずに理解できるはずである．関数型言語，論理型言語，代数型言語などといった範疇に属するプログラム言語は，この要請を満たしている*．

ML などの関数型言語のプログラムは，数学的領域間の関数という静的なものを定義していると考えられる．そして，計算は，関数の定義を展開することにより行われる．Prolog などの論理型言語のプログラムは，論理式の集まりとして与えられる．そして，計算は，導出原理と単一化により行われる．OBJ 3 などの代数型言語のプログラムは，代数的構造という静的なものを定義していると考えられる．そして計算は，代数の定義に現れる等式による項書き換えにより行われる．

もちろん，関数型言語のプログラムは関数を定義しているといっても，プログラムはあくまでもテキストである．プログラムテキストからそれが意味している数学的な関数を得る方法を用意する必要がある．論理型，代数型言語に関しても同様である．この，プログラムに対し静的な意味を数学的に定義する理論は，プログラムの表示的意味論と呼ばれている．表示的意味論では，不動点は重要な役割を果たしている．**2.** では，関数型言語について，階乗を計算するプログラムを例に，そのことについて述べる．

プログラムの意味を考えるときには，プログラムで扱うデータタイプに対しても明確な意味を与える必要がある．プログラミングに有用なデータタイプとしては，リスト，ツリーなどの再帰的な構造をもつものがあるが，これらの意味は，一種の不動点として与えることができる．最小不動点だけではなく最大不動点も考えれば，無限リストなど，無限データ構造の意味も扱うことが可能になる．**3.** では，リストを例に，再帰的なデータタイプの意味について述べる．データタイプの意味を不動点で与えるためには，不動点の概念を一般のカテゴリに拡張する必要がある．カテゴリの言葉を用いると，初学者にはとっつきにくくなるかもしれないが，リストを例に平易な解説を試みる．

† Fixed Points and the Theory of Programming Languages by Hideki TSUIKI (Faculty of Environmental Information, Keio University).

†† 慶應大学環境情報学部

* 手続き的な言語に対しても，実行過程と独立した静的な意味を考えることができるが，かなり繁雑なものとなってしまう．

2. 再帰プログラムと不動点

2.1 関数型プログラムの操作的意味と表示的意味

関数型言語では，関数を定義することによりプログラムを行う．関数は，いくつかの関数を組み合わせることにより定義する．すでに定義されている関数を組み合わせるだけでは繰り返しの構造は定義できないので，関数の定義の中で自分自身を呼び出すことも許す必要がある．このようにして定義されたプログラムを，再帰プログラムと呼ぶ．たとえば，次のプログラムは階乗を計算するint 型から int 型への再帰プログラムである．

fun fact(x)=if x=0 then 1 else x*fact(x−1);

このプログラムは，「fact(x) を計算するには，x が 0 のときは 1 を返し，それ以外のときは，x と fact(x−1) を掛けなさい」と fact の計算方法を説明しているように読める．実際，プログラムの実行は，定義に基づいて関数を展開することにより行われる．たとえば，fact (3) は以下のように計算される．

```
fact (3)→if 3=0 then 1 else 3*fact(3−1)
        →3*fact(2)
        →3*(if 2=0 then 1 else 2*fact(2−1))
        →3*(2*fact(1))
        →3*(2*(if 1=0 then 1 else 1*fact
          (1−1)))
        →3*(2*(1*fact(0)))
        →3*(2*(1*(if 0=0 then 1 else 0*fact
          (0−1))))
        →3*(2*(1*1))
        →6
```

この計算過程は，スタックを用いると容易に計算機上にインプリメント可能である．このように，プログラムを実行するための計算規則によりプログラムの意味を与えることを，操作的意味論とよぶ．

一方で，fact プログラムをただ単に左辺と右辺が等しいと述べているとみると，プログラムの表現している数学的な関数 *fact* の満たすべき等式を書いていると考えることもできる☆．

$fact(0)=1$

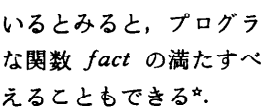
$fact(x)=fact(x-1)*x \quad (x\neq 0)$

と展開して書いたほうが分かりやすいかもしれない．これは，上のプログラムに計算方法によらない静的な意味を与えているとみることができる．すなわち，「*fact* とは，この等式を満たす関数」と *fact* を定義しているとみることができるわけである．このような，プログラムの実行方法によらず，数学的な対象（この場合には，整数の集合から整数の集合への関数）としてプログラムの意味を与えることを表示的意味論とよぶ．

fact の表示的意味を考えるとき，「この等式を満たす関数」と言っただけでは，何を定義しているのか意味がはっきりとしない．第一，この等式を満たすような整数から整数への関数が一つしかないという保証もないし，全ての引数に対して矛盾なく定義されるとも限らない．実際，この *fact* の定義では，負の数に対してどのような値を割り当てても矛盾を来たしてしまう．そこで，表示的意味を考えるのに，不動点という数学的な道具が必要となる．

2.2 表示的意味のための領域

fact(−1) を実行してみると，fact(−2), fact(−3) と次々と呼び出して，計算が止まらないことが分かる．プログラムの意味は，できるだけ実行時の動作を忠実に表現していることが望ましい．fact の意味も，単に整数から整数への関数ではなくこの「止まらない」ということも，表現する必要がある．そのために，関数の引数および結果の型に対応する領域として，「定義されていない」あるいは「実行したとき止まらない」ということを示す値ボトム(⊥)を付け加えた領域を考える．たとえば，整数の集合 *int* の代わりに，整数の集合に整数の型のボトム $\perp_i$ を加えた領域 $int_\perp$ を考える．$\perp_i$ は，計算しようとしても値が求まらないので，値について何の情報もなく，値のもつ情報量で順序を付けると，どの値よりも小さいと考えることができる．すなわち，$\perp_i<0, \perp_i<1, \cdots$ である．



この順序は，整数の大小関係とはまったく関係ないことを注意されたい．この順序により，$int_\perp$ は，部分順序集合 (partially ordered set)，特に，

☆ ここでは，プログラムは roman，プログラムの意味する数学的な関数は *italic* とフォントを区別して使うことにする．

完全部分順序集合 (complete partially ordered set, 以下, cpo と略する) をなしている. このように, 一般に, 型の表す領域は cpo と考える*.

プログラム fact の意味は, $int_\perp$ から $int_\perp$ への以下の関数 $fact_\perp$ である.

$fact_\perp(x)=fact(x)$ $(x\geq 0)$

$fact_\perp(x)=\perp_i$ $(x<0)$

$fact_\perp(\perp_i)=\perp_i$

ここで, $\perp_i$ に対する値は $\perp_i$ としている. これは, $fact(n)$ の値を求めるのには, n の値を求める必要があり, n の値が求まらないときには, $fact(n)$ の値も求まらないので自然である. ここで, $\perp_i$ は, 実行時の意味としては「止まらない」ということを意味しているが, 表示的意味論の世界では, $int_\perp$ の一つの要素に過ぎないことに注意して欲しい. すなわち, $fact_\perp(-1)$ は, $\perp_i$ という一つの値である. こうして cpo 間に拡張された関数は, 順序を保存しており, 連続関数である**. 一般に, 関数型言語のプログラムは, cpo 間の連続関数を表現していると考える.

ところで, cpo A から cpo B への連続関数全体も cpo をなすことが分かる. この cpo を $[A\rightarrow B]$ と書くことにする. $[A\rightarrow B]$ で順序は, $f(x)<g(x)$ が全ての x について成り立つとき $f<g$ と定義する. これを $[int_\perp\rightarrow int_\perp]$ に当てはめて考えれば, $f<g$ というのは, g のほうがより多くの引数に対して定義されていて, $f(a)$ と $g(a)$ が両方とも定義されている場合には等しいということを意味している. すなわち, 同じ関数だが, f より g のほうが定義域が広いということである. また, $[int_\perp\rightarrow int_\perp]$ の最小元 ($\perp_f$ と書くことにする) は, 全ての引数に対して $\perp_i$ を返す関数である.

関数型言語では, 型Aから型Bへの関数の型A->Bが存在し, 関数も値として扱う. これにより, 関数を引数としたり結果とする関数 (汎関数) も表現可能である. 型A->Bが表す領域は, A, Bが表す領域を A, B としたとき, $[A\rightarrow B]$ という cpo である. これにより, プログラムの意味は, 関数型 (の表す cpo) の要素として考えられる. たとえば, $fact_\perp$ は, $[int_\perp\rightarrow int_\perp]$ の要素である.

* cpo の定義は省略する. ここでは cpo としたが, 型の意味として, 計算可能などの性質を表現するため, もっと複雑な構造を考えることが多い.

** cpo 間の連続関数の定義も省略する.

2.3 再帰的関数の表示的意味

さて, 元に戻って, 再帰的プログラム fact からいかにしてその意味する関数 $fact_\perp$ を求めることができるか考えよう. fact のプログラムで右辺の fact を変数だと思うと, 古い fact をもらって新しい fact を返す汎関数が考えられる. すなわち,

```
val FACT=fn f=> fn x=> (if x=0 then
    1 else x*f(x-1));
```

という関数をもらって関数を返すプログラムを考える. (右辺の fn f=>… は, f をもらって … を返す関数を表している. また, val FACT=… というのは, … によって定義されるものを変数 FACT の値とするということである.) FACT に現れる *, -, =, if_then_else の意味として, ボトムを保存するように拡張した関数を考える. すなわち, $*_\perp$, $-_\perp$ を, それぞれ両方の引数が int に属するときには, *, - と等しく, どちらかの引数が $\perp_i$ なら $\perp_i$ を返す $[int_\perp\times int_\perp\rightarrow int_\perp]$ の関数, $=_\perp$ を両方の引数が int のときには = と等しくどちらかの引数が $\perp_i$ なら $\perp_b$ を返す $[int_\perp\times int_\perp\rightarrow bool_\perp]$ の関数とする. (ここで, $bool_\perp$ は, $true$, $false$, $\perp_b$ の三つの値をもつ領域である.) また, $if_\perp\_then\_else$ を, 第1引数が $\perp_b$ なら $\perp_i$, $true$ なら第2引数, $false$ なら第3引数の値を返す $[bool_\perp\times int_\perp\times int_\perp\rightarrow int_\perp]$ の関数とする. プログラム FACT は再帰的ではないので, このプログラムの意味 $FACT$ は, $[[int_\perp\rightarrow int_\perp]\rightarrow[int_\perp\rightarrow int_\perp]]$ の要素として明確に与えられる. $FACT$ を言葉で説明すると, f という $[int_\perp\rightarrow int_\perp]$ に属する関数をもらって, 「$int_\perp$ の要素 x をもらって x が $\perp_i$ なら $\perp_i$ を, 0なら1を, そうでないなら $x*_\perp f(x-_\perp 1)$ を返す関数」を返す関数である. $FACT$ が連続関数であることも, 容易に確かめられる.

さて, $FACT(fact_\perp)$ は, 「$int_\perp$ の要素 x をもらって x が $\perp_i$ なら $\perp_i$ を, 0なら1を, そうでないなら $x*_\perp fact_\perp(x-_\perp 1)$ を返す関数」である. ここで, $x>0$ のときには $x*_\perp fact_\perp(x-_\perp 1)=x*_\perp fact_\perp(x-1)=x*_\perp fact(x-1)=fact(x)$, $x<0$ のときには $x*_\perp fact_\perp(x-_\perp 1)=x*_\perp fact_\perp(x-1)=x*_\perp\perp_i=\perp_i$ である. よって, $FACT(fact_\perp)$ は $fact_\perp$ に等しい. すなわち, $fact_\perp$ は $FACT(X)=X$ を満たしており, 関数 $FACT$ の不動点

であることが分かる．実際，$fact_\perp$ は，$FACT$ の唯一の不動点である．

このように，再帰的プログラムの意味は，プログラムの右辺に現れる定義される関数名を変数に変えて作られる汎関数を考え，その最小不動点として与えることにする．もちろん，こうして与えた表示的意味が，プログラムの操作的意味と合致することを言わなければならない．しかし，本稿では，プログラムの操作的意味は正確に与えておらず，ここでは省略する．不動点が存在することは，Tarski の不動点定理により保証されている．

定理 1：(Tarski の不動点定理) *cpo* 間の連続関数 F は不動点をもつ．*cpo* の最小元を $\perp$ とすると，$\perp$, $F(\perp)$, $F^2(\perp)$, $F^3(\perp)$ という列は，上昇系列である．この上限は，最小不動点と一致する．ここで $F^2(\perp)$ は，$F(F(\perp))$ の略である． □

$FACT\ (\perp_f)$, $FACT^2(\perp_f)$, $FACT^3(\perp_f)$ を実際に計算してみれば，$\perp_f < FACT(\perp_f) < FACT^2(\perp_f)$ で，その上限が $fact_\perp$ と一致することが理解できるだろう．

$$
\begin{aligned}
\perp_f(x) &= \perp_i \\
FACT(\perp_f)(x) &= 1 \quad if\ (x=0) \\
&= \perp_i \quad else \\
FACT^2(\perp_f)(x) &= 1 \quad if\ (x=0) \\
&= 1 \quad if\ (x=1) \\
&= \perp_i \quad else \\
FACT^3(\perp_f)(x) &= 1 \quad if\ (x=0,1) \\
&= 2 \quad if\ (x=2) \\
&= \perp_i \quad else \\
FACT^n(\perp_f)(x) &= fact(x) \quad if\ (x<n) \\
&= \perp_i \quad else
\end{aligned}
$$

すなわち，$FACT^n(\perp_f)$ は，n より小さい引数に対してのみ計算を行う $fact$ 関数ということになる．これは，$fact_\perp$ に対する有限の近似である．雑な言い方をすれば，それらの近似を上限を取ることにより取りまとめたものとして $fact_\perp$ は定義される．

$FACT$ の場合には不動点は $fact_\perp$ だけしかなく，どの不動点を取るかという問題は生じない．しかし，一般には不動点は複雑存在し，なぜ再帰的関数の意味として最小不動点を取るのかという疑問が残るであろう．これは，一つには，与えられたプログラムを実行したときに「止まるかどうか」という意味と，最小不動点が一致することがある．また，関数の再帰的な定義が与えられたときに，その定義を満たす最小の関数，すなわち，その定義から導かれる情報だけで得られる関数を定義しているとみるのが自然だからとも説明できる．

たとえば，

```
fun f(x)=if x=0 then 1 else f(x+1);
```

という再帰的プログラムを考える．これに対応する汎関数

```
val F=fn f=>fn x=>if x=0 then 1 else
  f(x+1);
```

を考え，その意味する関数を F とすると，

$f(x)=k \quad (x>0)$

$f(x)=1 \quad else$

という関数 $f \in [int_\perp \to int_\perp]$ は，それぞれの k $(k \in int_\perp)$ に対して，F の不動点となっている．その中で最小不動点は，$k=\perp_i$ の場合である．一方，このプログラムを実行すると，負の数および 0 に対しては 1 を返し，それ以外に対しては止まらない．よって，このプログラムの意味する関数は，$x>0$ に対して $\perp_i$ を返しているとみることができる．また，プログラムから得られる f の満たす等式は

$f(0)=0$

$f(x)=f(x+1)$

であり，この等式を満たすということからは $f(x)$ $(x>0)$ の値は導けない．よって，$f(x)$ $(x>0)$ は定義されていない $(\perp_i)$ と考えるのが自然である．

2.4 その他の言語の意味論と不動点

ここで述べた最小不動点による意味論と同様の方法は，論理プログラムの意味を考えるときにも用いられる．論理プログラムではプログラムは論理式の集合である．そして，その Herbrand モデルとは，論理式を満足するようにそれぞれの ground atom に真偽値を割り振ったものである．プログラムが与えられたとき，そのモデルは幾つも存在するが，その中でプログラムの実行と対応するのは，真となる atom が，プログラムから論理的帰結として導かれるものだけからなるものである．これは，least Herbrand model と呼ばれ，真値集合をプログラムに基づいてどんどん増やしていく関数の最小不動点と一致する．この最小不動点の存在も，Tarski の不動点定理により保証される．

また，代数的言語の意味は，initial algebra として与えるが，これも最小不動点に似た考え方である．実際，term algebra が initial algebra となっているが，その構成は，ある関数の不動点として定義することもできる．

これらに共通する考え方を述べるとこうなる（括弧の中は，関数型言語の場合の例である）．プログラム（fact プログラム）は，ある数学的対象（$int_\perp$ から $int_\perp$ への $fact_\perp$ 関数）を意味している．その数学的対象の属する領域（$[int_\perp \to int_\perp]$）をドメインと呼ぶことにする．プログラムの記述から，そのドメインからそのドメインへの連続関数（$FACT$）が構成される．そして，その最小不動点が求める数学的対象と一致する．最小であることは，プログラムの記述から導かれる以外の余分な情報をもっていないことを意味する．

2.5 不動点オペレータ

再帰的プログラムは，関数を展開して実行することを考えれば自然であるが，フォーマルに意味を考えようとすると，再帰的でないプログラムを考えその不動点を取る必要があり，直接的ではなかった．実際，fact というプログラムを書くのに，これから定義するはずの fact を用いるというのは，すでに存在するものを用いて新しいものを作っていくという構成的な態度に反している．

もし，おのおのの型 σ に対して，FIX-σ という，$(\sigma\to\sigma)\to\sigma$ という型に属するオペレータで，関数に対してその最小不動点を返すものがプログラム言語に存在すれば，再帰法は必要がなくなる．

たとえば，fact 関数は，

```
FIX−{int−>int}(fn f=>fn x=>(if x
  =0 then 1 else x*f(x−1)))
```

と表現できる．再帰法を用いるときには，右辺から定義される関数を呼び出すために，定義される関数に名前 (fact) を付けてやらなければならなかったが，ここではその必要がないことに注意されたい．

FIX-σ の表示的意味は，型 σ に対応する意味領域を $\sigma_\perp$ としたとき，$\sigma_\perp$ から $\sigma_\perp$ への連続関数をもらってその最小不動点を返す $[\sigma_\perp\to\sigma_\perp]$ から $\sigma_\perp$ への連続関数である．この関数を，FIX_σ と書くことにしよう．

FIX-σ を用いて書かれたプログラムの計算規則（すなわち，操作的意味）は，次のルールとして与える．

FIX-σ(M)→M(FIX-σ(M))

これを用いると，たとえば，上の fact 関数 (FIX−{int−>int} を FIX, fn f=>fn x=>… を FACT と略記する）の適用は，

```
FIX(FACT)(3)→FACT(FIX(FACT))(3)
          →if 3=0 then 1 else 3*FIX
            (FACT)(3−1)
          →3*FIX(FACT)(2)
          …
          →6
```

と計算が行われる．これは，この章の最初に書いた，再帰呼び出しを展開して計算する方法と同じである．この FIX-σ は，不動点オペレータと呼ばれている．関数型言語に関して，新しい計算規則を考えたり，プログラムの性質を調べたりといった，基礎理論を考える場合には，再帰呼び出しの代わりに FIX-σ を用い，その基礎理論に基づく実際的な言語を設計するときには，書きやすさのために，再帰的な書き方を導入するというのが普通である．実際，本稿はプログラムの例を与えるのに ML[6] という関数型言語を用いているが，ML では，非再帰的関数を定義するには f=fn x ⇒…，再帰的関数を定義するには fun f (x)=…f …という表記を用いている．前者は＝の右辺に定義されたものを f に代入しているのに対し，後者は関数定義の中で定義される関数を参照するための特別な構文である．

関数に対してその不動点を取るという関数は，普通の型の概念のある言語では，ユーザがプログラムとして表現することはできない．それゆえ，FIX-σ というオペレータを用意する必要があった．それに対して，λ カリキュラスという，型の概念のない計算体系では，一つのプログラムとして，不動点関数を表現することができる．

λ カリキュラスは，関数の定義と適用の概念を抽象化した計算の体系であり，それ自身プログラム言語としてみることもできる．λ カリキュラスでは，型という概念はなく，全ての式は，値としても，値全体を引数，結果の領域とする関数としても扱うことができる．λ カリキュラスの式は，x を変数，e, e_1, e_2 を式としたとき

$e ::= x \mid \lambda x.e \mid e_1\ e_2$

と定義できる．ここで，$\lambda x.e$ は，e という x を

含む式に対して，x を引数として e を返す関数を意味している．また，$e_1\ e_2$ は，e_1 という関数を e_2 という引数に適用することを意味している．よって，$(\lambda x.e)e_1$ という式は，e において x の出現を e_1 に置き換えた式に書き換えられる．この書き換えによって，計算は行われる．

λ カリキュラスでは，不動点関数が，一つの式として表現可能である．

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

というのが最小不動点関数である．実際，書き換えを行うと，

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

となり，確かに $Y(f)$ が f の不動点となることが分かる．λ カリキュラスは，不動点が表現可能なことより，非常に単純な体系にもかかわらず，計算機で計算可能な関数が全て表現可能であり，十分強力であることが分かる．

$Y(f)$ は意味を考えたときに（他の不動点ではなく）本当に最小不動点となるのかということが問題になる．そのためには，λ カリキュラスの式を解釈する領域を構成する必要がある．λ カリキュラスでは，全ての値が関数としても扱うことができるため，意味を考えるには，$D=[D\rightarrow D]$ という関数を満たすドメイン D が必要である．このような D 上で，Y が，最小不動点を与えているということが証明される．詳しくは，文献 12) などを参照されたい．

2.6 不動点とプログラムの検証

表示的意味論により，プログラムの意味を静的な数学的対象として与えることを説明した．表示的意味が与えられると，二つのプログラムが等しい意味をもつとか，プログラムが仕様を満たしているといった，プログラムの性質を論じることが可能になる．また，コンパイラやプログラム変換がプログラムの意味を変えないということも，厳密に議論することが可能となる．

不動点として意味が与えられた関数の性質は，不動点帰納法という推論規則を用いて証明される．

$A(F)$ を意味領域 σ 上の述語とし，今，$A(F)$ が，$F:\sigma\rightarrow\sigma$ の不動点 $FIX_\sigma(F)$ に対して成り立つことを証明したいとする．不動点帰納法は，

1. $A(F)$ は許容的である．
2. σ の最小元 $\perp_\sigma$ に対し，$A(\perp_\sigma)$ が成り立つ．
3. 全ての $f\in\sigma$ に関し，$A(f)$ ならば $A(F(f))$ が成り立つ．

ということから，$A(FIX_\sigma(F))$ が成り立つことを推論する．ここで，許容的とは，$y_0\leq y_1\leq y_2\leq\cdots$ という列に対し，$A(y_i)$ $(i=0,1,2,\cdots)$ が成り立つなら，y_i の上限 $\bigsqcup_i y_i$ に対しても，$A(\bigsqcup y_i)$ が成り立つということである．不動点帰納法の推論の正当性は，$FIX\sigma(F)=\bigsqcup_i F^i(\perp_\sigma)$ であることから明らかである．

例をあげて説明しよう．

```
val F=fn f=>fn x=>if x>100 then x-10
  else f(f(x+11));
```

とし，FIX(F) が

```
val G=fn x=>if x>100 then x-10 else 91;
```

に対し，意味領域上で $FIX_{[int\rightarrow int]}(F)<G$ である（すなわち，FIX(F) の実行は，もし止まればGと同じ結果を出す.）ことを証明しよう．ここで，F,G は，それぞれ F,G の意味関数である．ここで，FIX－{int－>int} を，FIX と略記した．FIX(F) は，マッカーシーの 91 関数と呼ばれている．

証明：$A(f)$ を，$f<G$ とおく．1番の条件は上限の性質から明らかである．2番の条件は，明らかである．3番の条件は，$f<G$ のとき，$F(f)<G$ を示せば良いが，読者のほうに任せたい．

表示的意味論に基づいた論理を用いてプログラムの性質の証明を行うシステムに，LCF がある[10]．LCF では，ドメイン（表示的意味論の意味領域として現れた cpo）上での一階述語論理 (PPλ) を論理として採用しており，推論規則に不動点帰納法をもつ．よって，上にあげたような証明を形式的に行うことが可能である．LCF 以降，多くの定理証明システムが作られたが，このようなシステムの論理は，ドメイン上の論理に代わって，Martin_Lof の型論理，高階型論理 などが主流になりつつある．

3. データ・タイプと不動点

プログラムを書くときには，型として，int, real などの基本型，積型，関数型などの構造型のほかに，リストなどの再帰的構造をもった型を扱えれば便利である．型 A のリストとは，A の型の要素を並べたものである．たとえば，(1 2 3), (2),

( ) などは，型 int のリストである．( ) は空リストであり，nil とも書く．型 A のリスト全体のなす集合のことを，$LIST(A)$ と書くことにする．型 A の要素 a と型 $LIST(A)$ の要素 l から，a を l の先頭に付け加えることにより，$LIST(A)$ の要素 $cons(a,l)$ を構成できる．また逆に，$LIST(A)$ の要素 l は，空リストであるか，あるいは，型 A の先頭の要素 $car(l)$ と型 $LIST(A)$ に属する残りのリスト $cdr(l)$ に分解できるかどちらかである．よって，$LIST(A)$ は，

$$LIST(A)=A*LIST(A)+nil$$

という等式を満たしている．ここで，$+$, $*$は それぞれ集合の和と積を表していると考える．この等式は，右から左へは cons, 左から右へは<car, cdr> というリストに対する基本的なオペレーションを表しており，リストの基本的構造を表現している．すなわち，リストという型は，この等式によって定義されると考えられる．ここでは，このような等式が与えられたとして，その等式が意味するデータ型を得る方法を考える．

さて，型はデータの集合を表していると考えられる．$LIST(A)$ の表す集合は，X という集合をもらって $A*X+\{nil\}$ という集合を返す，集合の上の関数を F としたとき，$X=F(X)$ という等式を満たす集合と定義するのが自然であろう．この等式を満たす集合は複数存在するので，その中でも，最小のものと考えるのが自然なように思える．

しかし，これは，そう簡単ではない．集合と集合を比較しているのだから，＝は同型という意味に取るのが自然だろう．すると，X が無限集合なら，X と $F(X)$ は濃度が等しいので常に同型である．よって，全ての無限集合がこの関数の不動点となってしまう．そして，その中で最小のものを考えようとしても，無限集合には無限集合が部分集合として含まれるので，いくらでも小さな集合が存在することになる．

この不具合は，＝の意味を任意の同型としたことによる．上の例で，＝は cons, 〈car, cdr〉 という意味をもっている．この等式により，$LIST(A)$ に代数的な構造が作られていると考えられる．そこで，最小の意味を，同じ代数的構造を保つような部分集合が存在しないという意味で考えればよさそうである．すなわち，集合 X で $F(X)$ と X が同型で，その同型写像を ϕ とすれば，X の真部分集合 Y で $F(Y)$ と Y が同じ ϕ で同型となるものは存在しない．そのような集合を $LIST(A)$ とすると．このことをより厳密に表現するには，カテゴリの言葉を用いるのが便利である．以後カテゴリの基本的な用語は断わりなく用いる．詳しくは文献 3) を参照されたい．なお，テクニカルな理由で，カテゴリで考えるときには，$F(X)$ と X が ϕ により同型になる場合ではなく，単に $F(X)$ から X に射 ϕ が存在する場合を扱う．

定義1：Cをカテゴリ，F を CからC へのファンクタとする．オブジェクト X と，$F(X)\to X$ の射 ϕ のペア (X,ϕ) のことを，F-代数 (F-algebra) と呼ぶ．(X,ϕ), (Y,ϕ') が F-代数のとき，$f: X\to Y$ で，以下の図式を可換にするものを (X,ϕ) から (Y,ϕ') への F-代数間の射と定義する．

$$\begin{array}{ccc} F(X) & \xrightarrow{\phi} & X \\ \downarrow{\scriptstyle F(f)} & & \downarrow{\scriptstyle f} \\ F(Y) & \xrightarrow{\phi'} & Y \end{array}$$

すると，F-代数全体はカテゴリをなす．このカテゴリの始対象のことを，F の最小不動点と定義する．□

(X,ϕ) が F-代数のなすカテゴリの始対象のとき，ϕ が $F(X)$ から X への同型射になるということが，一般にいえる．これは，X が F の不動点であることを意味している．また，集合のカテゴリ Set で考え，(X,ϕ) が F-代数のなすカテゴリの始対象とし，Y を X の部分集合で，ϕ により $F(Y)$ と Y が同型となるものとする．すると，(X,ϕ) が始対象であることより，射 $f: X\to Y$ が存在し，$f\circ i=id_X$ となる．ここで，i は Y の X への埋め込みである．よって，i が全単射となり，Y と X は同型となる．すなわち，X の真部分集合で，ϕ により $F(Y)$ と Y が同型となるものは存在しない．よって，カテゴリによる最小不動点の定義は，上にインフォーマルに述べた最小不動点の定義を包含している．

cpo において，$a<b$ のとき a から b に射があると定義すると，cpo は一つのカテゴリとみなせる．カテゴリの最小不動点の定義は，cpo における最小不動点を包含していることが分かる．また，cpo における Tarski の不動点定理を一般のカテゴリに拡張したものが存在する．

定理2：カテゴリCが始対象 $\perp$ をもち，

δ: $\perp \xrightarrow{l} F(\perp) \xrightarrow{F(l)} F^2(\perp) \xrightarrow{F^2(l)} F^3(\perp)\cdots$

という列（ここで l は，$\perp$ から $F(\perp)$ に唯一存在する射）が逆極限 (colimit) をもち，その逆極限が F によって保存されるなら，F は最小不動点 (X, ϕ) をもつ．X は δ の逆極限であり，ϕ: $F(X)\to X$ は，$F(\delta)$ から，δ への列間の恒等射 $F^n(\perp)\to F^n(\perp)$ の逆極限である．□

任意の逆極限を保存するというのは，cpo の場合，F が連続写像ということに対応する．

さて，Set の場合に戻って，この意味を考えてみる．Set において，列 δ の逆極限とは，$F^n(\perp)$ の和集合において，$F^n(l)$ によって移り合う要素を同一視したものである．X という集合をもらって $A*X+\{nil\}$ という集合を返す Set から Set へのファンクタを F と置く．Set の場合には，$*$ と＋で表されたファンクタは，全ての逆極限を保存する．よって，そのようなファンクタ，特に F は，最小不動点 $(LIST(A), \phi)$ をもつ．

Set の場合には，$\perp$＝空集合である．δ による構成をみると，逆極限が A のリストの集合であることがよく分かる．

$\perp$　　：空集合

$F(\perp)=A*\perp+\{nil\}=\{nil\}$ ：
長さ 0 のリストの集合

$F^2(\perp)=A*\{nil\}+\{nil\}=A+\{nil\}$ ：
長さ高々 1 のリストの集合

$F^3(\perp)=A*(A+\{nil\})+\{nil\}$ ：
長さ高々 2 のリストの集合

である．$F^n(l)$ は $F^n(\perp)$ から $F^{n+1}(\perp)$ への自然な埋め込みであり，逆極限はこの埋め込みで移り合うものを同一視した $F^n(\perp)$, $(n=1,2,\cdots)$ の和集合，すなわち，長さ 1, 2, 3, …のリストを全て集めてきた集合を意味する．

最小不動点を与えるのに F-代数のカテゴリで始対象を考えたが，双対概念として最大不動点も定義できる．これは，F-双対代数 (F-coalgebra) のカテゴリを考え，そこで終対象を考えればよく，C の終対象 T に対し，

δ': $\top \xleftarrow{t} F(\top) \xleftarrow{F(t)} F^2(\top) \xleftarrow{F^2(t)} F^3(\top)\cdots$

という列の極限 (limit) と等しくなる．

Set の場合で，上と同じ F に対して計算してみると，

T　　：一元集合

$F(\top)=A*\top+\{nil\}=A+\{nil\}$ ：
長さ高々 1 のリストの集合

$F^2(\top)=A*(A+\{nil\})+\{nil\}$ ：
長さ高々 2 のリストの集合…

となり，$F^n(\top)$ は，$F^{n+1}(\perp)$ と同じ構造である．$F^{n+1}(\top)$ から $F^n(\top)$ への写像 $F^n(t)$ は，長さが n+1 なら最後の要素を無視するという写像となる．これの極限は，$F^n(\top)$ $(n=0,1,2,\cdots)$ の無限積の部分集合で，第 $n+1$ 成分と第 n 成分が $F^n(t)$ で互いに移るものの集合である．すなわち，$(a_1, a_1a_2, a_1a_2a_3, a_1a_2a_3a_4, \cdots)$ という無限列か，途中から同じものばかりが現れる $(a_1, a_1a_2, a_1a_2, a_1a_2, \cdots)$ といった無限列である．前者は $(a_1a_2a_3a_4, \cdots)$ という無限リスト，後者は (a_1a_2) という有限リストを表している．よって，この最大不動点は，型 A の有限，あるいは無限リスト全体の集合である．

無限リストは，ストリームとも呼ばれ，lazy な計算をもつ関数型言語で使われる．リストと無限リストが，それぞれ同じファンクタの最小，最大不動点として表現できることは，興味深い．ファンクタの最小不動点となる型をユーザが定義可能な言語は多いが，最大不動点が定義可能な言語は少ない．文献 13) は，最大，最小不動点だけではなく，より一般に，積や和，関数空間などの型もユーザがカテゴリを用いて定義可能な言語である．

ここではデータ型を集合として説明したが，**3.** でも扱ったように，データ型は，cpo と考える必要がある．cpo のカテゴリにおいても，$*$ と＋で表されたファンクタは，全て逆極限を保存するので，リストなどの型を不動点として定義することができる．

λ カリキュラスでは，全ての式が関数としても値としても扱われる．特に，関数を自分自身に適用することも可能である．このような言語の意味を考えるためには，

$D=[D\to D]$

といった領域方程式を解く必要がある．しかし，$F(X)=[X\to X]$ という cpo 間の写像は，ファンクタではないために，リストのときの手法はそのままでは使えない．これを解決するためには，射として連続関数ではなく，embedding-projection ペアを取るカテゴリを考える．このカテゴリでは

Fはファンクタとなり，不動点を求めることができる．詳しくは，文献 2) を参照されたい．また，ドメインの等式を解くことを，cpo の上で連続関数の不動点を求めることに帰着する方法もある[5),7)].

4. おわりに

再帰プログラムの意味論と不動点，データタイプと不動点の関係について説明した．前半は，文献 5)，後半は文献 3),8) を主に参考にした．また，龍谷大学助教授の林晋先生，慶應義塾大学専任講師の萩野達也先生，および三好博之氏には草稿を読んでいただき，貴重な助言をいただいた．

ここで説明したように，cpo 上の Tarski の不動点定理，およびその拡張は，計算機科学で最も馴染みの深い不動点定理である．文献 4) は，オブジェクト指向に現れる継承の意味を関数型言語の不動点意味論と同様の方法で説明している．また，その他の不動点定理も計算機科学で有効に使われている．たとえば，文献 1) では，データ型の意味をイデアルとして解釈しているが，再帰的に定義された型に対応するイデアルが存在することを述べるのに，イデアル間の距離を定義し，距離空間における Banach の不動点定理を用いている．また，コンカレントプログラムの意味を与えるドメインを，距離空間上の不動点で構成する研究もなされている[9)]．

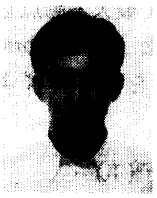
今のところ，プログラムの意味論は，プログラムを実際に書く人にとっては，知らなくてもプログラムが書けるが，知っていたほうがより見通しの良いプログラムを書くのに役立つといった程度の意味しかもっていないと思われる．しかし，ML などの関数型言語，Prolog などの論理型言語など，意味論的基礎をもつ言語が，手続き的なプログラム言語に代わって広く使われてきつつある．また，オブジェクト指向言語に対しても，最近，意味論的基礎を与えるような研究がなされている．このことは，意味論が，優れたプログラム言語を設計するのに役に立っているということを意味していよう．

これから，分散環境や並列環境を利用したプログラミングが行われるようになっていくだろうが，そのような，より複雑な計算機構を利用するためのプログラム言語の設計に，意味論は重要な役割を果たしていくだろう．

参考文献

1) MacQueen, D., Plotokin, G.D. and Sethi, R.: An Ideal Model for Recursive Polymorphic Types, Information and Control, 71, pp. 95-130 (1986).
2) Smyth, M.B. and Plotokin, G. D.: The Category-Theoretic Solution of Recursive Domain Equations, SIAM J. Comput, 11, 4 (1982).
3) Barr, M. and Wells, C.: Category Theory for Computing Science, Prentice Hall (1990).
4) Cook, W.R., Hill, W.L. and Canning, P.S.: Inheritance Is Not Subtyping, Proc. 17-th ACM Symp. on Principles of Programming Languages, pp. 125-135 (1990).
5) Gunter, C.A. and Scott, D.S.: Semantic Domains. Handbook of Theoretical Computer Science, Volume B, pp. 633-674, van Leeuwen, J. Eds, MIT Press (1990).
6) Milner, R., Tofte, M. and Harper, R.: The Definition of Standard ML, MIT Press (1990).
7) Larsen, K.G. and Winskel, G.: Using Infor-mation Systems to Solve Recursive Domain Equations, Information and Computation, 91, pp. 232-258 (1991).
8) Manes, E.: An Example in Program Semantics, U.S.-French Seminar on the Application of Algebra to Language Definition and Compilation, Fontainebleau, France (1982).
9) Bakker, J.W. and Zucker, J.I.: Process and the Denotational Semantics of Concurrency, Information and Control, 54, pp. 70-120 (1982).
10) Paulson, L.C.: Logic and Computation, Cambridge University Press (1987).
11) Stoy, J.E.: Denotational Semantics, MIT Press (1977).
12) 中島玲二：数理情報学入門，朝倉書店 (1982).
13) 萩野達也：カテゴリー理論的関数型プログラミング言語，コンピュータソフトウェア，Vol. 7, No. 1. pp. 16-32.

(平成3年7月2日受付)



立木 秀樹（正会員）

1963 年生．1986 年京都大学理学部卒業．1988年京都大学理学部大学院修士課程（数理解析専攻）修了．同年，京都大学数理解析研究所助手．1990 年より慶應義塾大学環境情報学部助手．型理論，数理論理学，カテゴリ理論などのプログラム言語の基礎理論，マルチリンガル環境等に興味を持つ．ソフトウェア科学会，ACM 各会員．