計算機科学と代数学

プログラム意味論と普遍代数学

蓮尾 一郎
東京大学 大学院情報理工学系研究科 コンピュータ科学専攻

2014.1

1　計算機科学とは？ 意味論とは？

代数学のさまざまな応用についての本特集において，本稿は**計算機科学**——そのうちとくに**意味論**——とよばれる分野に現れる代数学的構造と，その応用について紹介したい．多くの読者のもともとの興味は数学や，多少外れたとしても物理学などだろうから，まず，計算機科学（のうち意味論）とは何をめざす学問なのかを説明していく．

ところでいきなり余談から始める．自分と異なる研究分野の人と話をする際，まず最初にやるべきことは，相手が「何がやりたいのか，何をカッコいいと思うのか」をお互いに理解することである．しかし，そのような目的意識・美意識は本人にとってはすでに刷り込まれた当たり前のことであり，当たり前のことを言葉にして説明するのはとてもむずかしい．この場合の有効な手段は，インフォーマルに徹してともかく「気持ち」をたくさん話すことである．というのはプリンストン高等研究所の学際研究担当教授 Piet Hut さんの受け売りであり，彼は「IZAKAYA に行くといいね」と言うが本稿ではそうもいかない．何が言いたいのかというと，本稿が以下ややくだけ過ぎ感があることの言い訳である．

計算機科学と一口に言っても研究内容はアーキテクチャ，アルゴリズム，ハイパフォーマンス・コンピューティング，ユーザ・インタフェイスなどさまざまであるが，本稿で注目するのは

- 計算機——特にプログラム——の動作，ふるまいを数学のコトバでモデリングして，その本質の理解をめざす**意味論**（semantics）というトピックと，
- その応用，特に計算機システムの正しさを数学的に証明する**形式検証**（formal verification）への応用[1]

[1]「形式」とはおおむね「数学的」の意味.

である．

上のふたつの●において「計算機」を「自然界」にとり替えると，そのまま物理学の説明として読めそうなことに注意されたい．実際，われわれ意味論の研究者が日々行っていることは，たとえば惑星の運動を微分方程式で記述するかのような**数理モデリング**である．ただしもちろん，対象の違いはモデリング手法の違いとして現れる： 連続的な自然現象のモデリングには主に解析学的手法によるが，プログラムのそれには代数学的手法を用いるのである．

本稿ではプロセス代数という単純なプログラミング言語を例に，計算機科学における意味論の現場をかいま見てもらうことを目的にする．はじめに注意しておくと，数値計算によるシミュレーションなどの**計算科学**や，数式処理を計算機上で行う**計算機代数**などは本稿のテーマとは別の話である．期待された読者の方には申し訳ない．

2　計算機科学における「有限 vs. 無限」

もう少しインフォーマルな話を続けたい．問題は

- 計算機やプログラムのどのような特性ゆえに，代数学的手法が有効なのか？

というものであり，筆者の解答は

- 有限と無限のせめぎあい

である．この点を説明していく．

計算機の行う「計算」とは，足し算や掛け算と言うより「なんらかの手順に従った機械的な作業」と思った方が適切だ．このことは，計算機が生活のあらゆる場面において使われる現代においては明らかだろう．この手順を記述するのが**プログラム**と呼ばれる有限長の文字列であり，計算機はプログラムにしたがって動作し作業を行う．

本稿のテーマである「意味論」という言葉は，たとえば「自然言語の意味論」というようにも用いられる一般的な言葉だが，本稿の文脈では「プログラムの意味について数学的に考えよう」という意味である．ここで： 次のプログラムの意味とはなんだろうか？

```
int add (int x, int y) {
  return x + y;
}
```
(1)

「足し算だよね」と思った読者はすでに（単純化・抽象化を伴う）数理モデリングを行っており，すなわち意味論の実践者であると言えよう．というのは，多くの実装において `int` （整数）型は 32 ビットのデータ型であり，足し算

の結果が桁あふれするかもしれないのである．このように数理モデリングによって失われるディテールが存在する一方で，単純化された「意味」を用いてプログラムの性質を数学的に記述できるのは大きな利点である．このあたりの事情は,（大きさを持たない質点が登場したりする）物理学などでも同じであろう．

プログラムの意味について考えると，本節はじめの「有限 vs. 無限」の対比が直ちに現れる．次のプログラムを考えよう．

```
infinite (int x) {
  print x;
  infinite (x + 1 mod 2);
};
infinite (0);
```
(2)

関数 `infinite` は自分自身の**再帰呼び出し**を行うため，このプログラムを実行すると画面上に

$$0, 1, 0, 1, \ldots$$

と表示され続けるはずである．ここで重要なのは，計算機が動作し続けることにより**有限長**のプログラムが**無限長**の意味を生成することである．

ここで，大学 2 年くらいで習う（素朴）集合論で考えてみると，

- プログラムは文字の有限列．すなわち，文字の集合（**アルファベット**とよばれ，ふつう有限集合）を Σ とすると，プログラムは $\Sigma^* = \bigcup_{n\geq 0} \Sigma^n$ の元．
- プログラムの意味（または**ふるまい**）は何らかの集合 Γ の元の無限列，すなわち $\Gamma^{\mathbb{N}}$ の元．

濃度を比較すると $|\Sigma^*| = \aleph_0 < \aleph_1 \leq |\Gamma^{\mathbb{N}}|$ （ただし $|\Gamma| \geq 2$ とした）．この帰結として，

- 可能なふるまい全体のうち，あるプログラミング言語を用いて表現できるものはごくわずか

であることがわかる [2]．

計算機科学の本質はまさにこの点にある： つまり，**無限長のふるまいについて，有限長のプログラムを用いて記述し，考える**のである．たとえば，

> `infinite` を用いた上のプログラム (2) のふるまいに `0` が無限回現れるかどうか？

[2] プログラムで記述できるふるまいはある意味で**有理的**（rational）であると言える．ただしこれは循環的とは違う： 無限列 0, 1, 0, 0, 1, 0, 0, 0, 1, ... は（際限なく大きなメモリを仮定すれば）再帰プログラムで生成できる．

という問題を考えよう．これを調べるために，ふるまいとしての無限列$\mathtt{0},\mathtt{1},\mathtt{0},\mathtt{1},\ldots$を直接調べるわけにはいかない——無限列を尽くす前に我々の有限の寿命が先に尽きてしまう．しかしもともとのプログラム（すなわち無限列の有限的表現）からスタートして，その構造を使ってやれば，余帰納法（coinduction）という手法を用いた証明がものの数行で書けてしまう．

3　代数学とは——普遍代数学

有限長のプログラムのもつ無限長のふるまいについて説明してきた．ここでなぜ代数学が登場するのだろうか？

そもそも「代数学」という言葉が何を指すのかについて，読者のみなさんもそれぞれイメージをお持ちのことと思うし，本特集では代数学のさまざまな側面が紹介され応用されている．ここではひとつ大胆なスローガンをぶちあげてみよう．

代数学とは**構文と意味の分離**とみつけたり

この見方を説明するため，さまざまな種類の代数系を統一的に扱う**普遍代数学**（universal algebra）のほんのさわりを導入する．普遍代数学は歴史的には**論理学**の一分野であり，意味論で用いる「代数学」とは多くの場合普遍代数学を指す．

定義 1

- **代数シグニチャ**（*signature*）とは，集合の無限列$\Sigma = (\Sigma_0, \Sigma_1, \Sigma_2, \ldots)$のこと．各$n \in \mathbb{Z}_{\geq 0}$に対し，元$\sigma \in \Sigma_n$を$n$**項演算子**とよぶ．
- 代数シグニチャΣに対し，次のルールを有限回適用して得られる記号列をΣ**項**とよぶ．
  - **変数** $\mathtt{x},\mathtt{y},\ldots \in \mathbf{Var}$は$\Sigma$項．
  - $\sigma \in \Sigma_n$がn項演算子で，$t_1,\ldots,t_n$がそれぞれΣ項ならば，$\sigma(t_1,\ldots,t_n)$はΣ項．

  ただし変数の可算無限集合 **Var** を別に定めておくものとする．
- Σ上の**等式公理**とは，Σ項のペア(s,t)のことをいう．等式公理(s,t)を$s = t$とも書く．

例を挙げると，

$$\Sigma_{\mathrm{gp}} \;=\; \big(\{\mathtt{e}\},\{\mathtt{i}\},\{\mathtt{m}\},\emptyset,\emptyset,\ldots\big)$$

は代数シグニチャであり，

`m(e(),x) = x`　や　`m(i(x),x) = e()`

は Σ_{gp} 上の等式公理である．

上の定義によると，Σ 項を作るためには変数 $\mathtt{x}, \mathtt{y}, \ldots$ あるいは 0 項演算子を使った $\sigma()$ から始めて，これらを $n(\geq 1)$ 項演算子を使って順次組み合わせていくことになる．こうしてできる Σ 項は常に有限長の文字列であることに注意しておく．

上の例からは「群のことかなあ」という雰囲気が伝わってくるが，ポイントは**記号の意味をまだ考えない**ことである．すなわち，演算子 $\mathtt{m}$ にはまだ何の意味も定義されていないし，括弧 (や) もただの記号であり，さらには，記号 = はただの区切りの役割しか果たしていない．次の定義ではじめて意味が登場する．

定義 2　• **代数仕様** *(algebraic specification)* とは，代数シグニチャΣ と，Σ 上の等式公理の（任意の）集合 E のペア (Σ, E) のこと．

- (Σ, E) **代数**とは $\big(X, \big([\![\sigma]\!]\big)_{\sigma\in\Sigma}\big)$，ただし
  - X は空でない集合（**台集合**）
  - 各 n と各演算子 $\sigma \in \Sigma_n$ に対して，$[\![\sigma]\!]$ は $X^n \to X$ の型の関数（演算子 σ の**解釈**）

  であり，E に属する等式公理すべてを「満たす」もののことをいう [3].

つまり，群を例にとって考えると，

- 群の定義はある代数仕様 $(\Sigma_{\mathrm{gp}}, E_{\mathrm{gp}})$ によって与えられ，
- 具体的な群（対象群，一般線形群，$\cdots$）はそれぞれひとつの $(\Sigma_{\mathrm{gp}}, E_{\mathrm{gp}})$ 代数を与える，

というわけである [4].

以上の定義はまだるっこしいかもしれない．しかしここで行ったのは**計算機科学（と論理学）の基礎の基礎**[5]，すなわち，記号列を取り扱う**構文論**（syntax）と，記号列の意味を数学的に定義する**意味論**（semantics）との分離である．ここで用いた記号 $[\![-]\!]$ は，論理学・計算機科学において一般に「文字列の意味」を表すために用いられる．

以上では群や環などの代数的概念が本質的に構文論的であることをみてきた．このことは代数学的手法の一類型，すなわち**数式変形**の基礎をなす：(Σ, E) 代数の元を Σ 項を用いて書き表したのち，E に属する等式公理を用いて変形

[3]「満たす」の定義の概略を述べる．変数から X の元への割り当て（**付値**）$J\colon \mathbf{Var} \to X$ を定めると，項 t の**解釈** $[\![t]\!]_J \in X$ が帰納的に（小さな項から順々に）定義できる．等式公理 $s = t$ が**満たされる**とは，すべての付値 J に対して $[\![s]\!]_J = [\![t]\!]_J$ となることをいう．最後の $=$ はただの区切り記号 = でなく，X の元としての等しさであることに注意．

[4]正確には,「群の定義の一表現」が代数仕様 $(\Sigma_{\mathrm{gp}}, E_{\mathrm{gp}})$ に相当すると言うべきである：たとえば，適切に等式公理を選ぶことによって，2 項演算 $(-_1)^{-1} \cdot (-_2)$ のみを用いた表現も可能．

[5]そもそも計算機（Turing 機械）は 20 世紀初頭の論理学の隆盛の中から現れた．

していくのである．代数学での構文論の重要性の例は他にも，自由群や加群のテンソル積の構成——記号列の同値類を用いて記述される——に現れる．また，最近数学の多くの分野で用いられる**オペラド**（operad）の理論は，上記の普遍代数学の「いとこ」のような存在である．

話を戻して：計算機科学と代数学の関わりについてはすでに明らかであろう．すなわち，双方とも

有限長の記号列による表現を基礎としている

のである（前者においてはプログラム，後者においては数式つまり Σ 項）．

ところで，計算機科学と代数学の関わりをしめす余談としては次のようなものもある．そもそも英語の algebra という単語の語源はアラビア語の al jebr（ばらばらにしたものを組み合わせる）であり，その単語を最初に著書のタイトルに使ったのが 9 世紀のイスラム科学者 al-Khwārizmī であり，彼の名前がアルゴリズム algorithm の語源となった．

4　プロセス代数

本稿の残りでは Plotkin によるプロセス代数の構造的操作意味論を通して，「意味論の働く現場」を見てもらいたい．

並列システム論（concurrency theory）という分野がある．マルチコア・プロセッサからインターネットまで，計算機を並列実行するのは現在の流行である一方，並列システムのふるまいについて考えるのはとてもむずかしい——おおざっぱに言って，計算ユニット数の増加に対してシステム全体の複雑さは指数関数的に増加する．**プロセス代数**（process algebra）とは，並列システム論においてシステムを表現するのに用いられる単純な「プログラミング言語」の一群を指す．

多くのプロセス代数は定義 2 のような単純な代数シグニチャと同一視できる（ゆえにプロセス「代数」）[6]．次の代数シグニチャを考えよう．

$$\Sigma_{\mathrm{pa}} = \big(\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathbf{0}\}, \{!\}, \{\parallel, +\}, \emptyset, \emptyset, \dots\big)$$

これらの演算子の意味を正確に定義することがそもそもの問題なのであり，そのやり方を後で説明するのであるが，とりあえず「気持ち」を述べておこう．Σ_{pa} 項のそれぞれは並列システムをあらわす**プログラム**であり，

- $\mathsf{a}, \mathsf{b}, \mathsf{c}$ は**（原子）アクション**．便宜的に，「画面に a（または b, c）を出力」と考えよう．
- $\mathbf{0}$ は**停止**を表す．

[6] 一般にプログラミング言語を代数的に表現するには，**多種シグニチャ**（many-sorted signature）が必要である．

- $s \parallel t$ はシステム s, t の**並列実行**を表す.
- $s + t$ はシステム s, t の**非決定和**（どちらかを任意に選んで実行するが，選ばなれなかった片方については忘れてしまう）.
- $!s$ は s の**複製**（s のコピーを可算無限個作って，これらを並列に実行する）.

例として Σ_{pa} 項 $\mathtt{a} \parallel (\mathtt{b} + \mathtt{c})$ を考えると，その表すシステムは

- システム $\mathtt{a}$ と $\mathtt{b} + \mathtt{c}$ が並列実行しており，どちらが先にアクションを行うかわからない．前者が先の場合は $\mathtt{a}$ を表示し，後者が先の場合は $\mathtt{b}$ と $\mathtt{c}$ のどちらかが表示される（そしてもう片方については忘れてしまう）.
- 次に，上で $\mathtt{a}$ が表示された場合は，残りの $\mathtt{b} + \mathtt{c}$ がアクションを行う．逆も同様.

このように，並列システムのふるまい——すなわちプロセス代数のプログラムの意味——を考えると，並列実行の**競合**（race）による非決定性が不可避的に現れる．これを数学的に記述するためには，プログラム (1) の場合のように入力値から出力値への関数と考えたり，プログラム (2) の場合のように画面出力の無限列とするのではうまくいかない．そこで並列システム論では以下のような**ラベル付き遷移系**（labeled transition system）を用いることが多い.

$$[\![\mathtt{a} \parallel (\mathtt{b} + \mathtt{c})]\!] \;=\; \left[\;\; \begin{array}{c} \downarrow \\ \bullet \\ \mathtt{a} \swarrow \quad \mathtt{c} \searrow \\ \bullet \qquad \mathtt{b} \qquad \bullet \\ \mathtt{c} \searrow \quad \swarrow \mathtt{a} \\ \mathtt{b} \qquad \bullet \end{array} \;\;\right] \tag{3}$$

つまり，一番上の状態 $\bullet$ から始まって，アクション $\mathtt{a}, \mathtt{b}, \mathtt{c}$ を行いながら状態を遷移していくわけである．この遷移系は上記の「気持ち」を数学的に表現しており，このようにシステム実行を追いかけるスタイルの意味論を**操作意味論**（operational semantics）と呼ぶ.

さて問題は，上記 (3) のような操作意味論をどのようにして数学的に正確に定義するかであり，一つの答えが Plotkin の**構造的操作意味論**（structural operational semantics, SOS）である.

5　構造的操作意味論（SOS）

最初に SOS のレシピの概略を述べておこう.

1. プロセス代数の各演算子の意味を **SOS ルール**として定義する.

$$\frac{s \xrightarrow{\mathtt{x}} s'}{s \parallel t \xrightarrow{\mathtt{x}} s' \parallel t}\ (\parallel\mathrm{L}) \qquad \frac{t \xrightarrow{\mathtt{x}} t'}{s \parallel t \xrightarrow{\mathtt{x}} s \parallel t'}\ (\parallel\mathrm{R})$$

$$\frac{s \xrightarrow{\mathtt{x}} s'}{s + t \xrightarrow{\mathtt{x}} s'}\ (+\mathrm{L}) \qquad \frac{t \xrightarrow{\mathtt{x}} t'}{s + t \xrightarrow{\mathtt{x}} t'}\ (+\mathrm{R})$$

$$\frac{s \xrightarrow{\mathtt{x}} s'}{!\,s \xrightarrow{\mathtt{x}} s' \parallel !\,s}\ (!) \qquad \frac{}{\mathtt{x} \xrightarrow{\mathtt{x}} \mathbf{0}}\ (\textsc{AtomAct})$$

図 1: Σ_{pa} のための SOS ルール

2. 項の間の遷移関係 $s \xrightarrow{\mathtt{x}} t$ を，SOS ルールを用いて導出する．

3. 項を状態として，その間を上で導出された遷移関係でつないでえられる遷移系を意味として与える [7]．

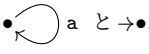
しばしば——具体的には SOS ルールの形が特定の**フォーマット**に従う場合——ステップ 2. の導出は項 s の構造にもとづき帰納的に行われる．これが「構造的」操作意味論と呼ばれるゆえんである．

さっそくステップ 1. から： 前節の Σ_{pa} に対して SOS ルールを与えよう．図 1 において，

- $\mathtt{x}$ は $\mathtt{a}, \mathtt{b}, \mathtt{c}$ のいずれかを表し，s, s', t, t' は任意の Σ_{pa} 項を表す．
- ルールの横に書いてある ($\parallel$L) や (AtomAct) はルールの名前である．「L」「R」は左，右を表す．
- ルールは上から下に読む（「横線の上の遷移がすでに導出されたならば，そこからさらに横線の下の遷移を導出してよい」）．つまり，上が仮定で下が結論である．最後の (AtomAct) ルールには仮定がなく，これを導出におけるベースケースとして用いることができる．

4 節に述べた各演算子の「気持ち」が，SOS ルールとして表現されていることを納得してもらえるだろうか．たとえば項 s から $\mathtt{x}$ 遷移が可能であるとき（すなわち，ある s' に対して $s \xrightarrow{\mathtt{x}} s'$），項 $s \parallel t$ と $s + t$ の両方から $\mathtt{x}$ 遷移が可能であるが，前者では t が生き残る一方で，後者では t は忘れられてしまう．

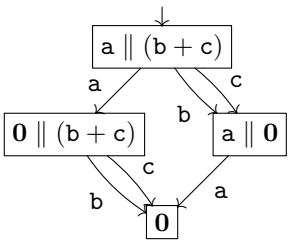
ステップ 2. に進もう．遷移関係は SOS ルールを有限回用いて導出する．たとえば項 $\mathtt{a} \parallel (\mathtt{b} + \mathtt{c})$ において，$\mathtt{b} + \mathtt{c}$ が先に動作して，しかも $\mathtt{b}$ が選ばれた場合には $\mathtt{b}$ 遷移がおこるが，これは次のように SOS ルールを用いて導出さ

[7] 本当は最後に「遷移系を**双模倣性**（bisimilarity）で割る」というステップがあるのだが，本稿では省略．双模倣性とは遷移系の間の同値関係の一つで，たとえば  と  を同一視する．

れる．

$$\dfrac{\dfrac{\dfrac{}{\mathtt{b} \xrightarrow{\mathtt{b}} \mathbf{0}}\ (\textsc{AtomAct})}{\mathtt{b}+\mathtt{c} \xrightarrow{\mathtt{b}} \mathbf{0}}\ (+\mathrm{L})}{\mathtt{a} \parallel (\mathtt{b}+\mathtt{c}) \xrightarrow{\mathtt{b}} \mathtt{a} \parallel \mathbf{0}}\ (\parallel \mathrm{R}) \tag{4}$$

項 $\mathtt{a} \parallel (\mathtt{b}+\mathtt{c})$ からはじめて，このような導出の可能なものをすべて挙げると次を得る．



(5)

最後にステップ 3. として，(5) において各状態に書いてある項を忘れると (3) の遷移系を得る．

図 1 の SOS ルールに対しては，(5) のように可能な遷移（すなわち，可能な導出）をすべて数え上げることはむずかしくない．なぜならば，各ルールにおいて仮定の左側は結論の左側の一部（**部分項**）になっているからである（たとえばルール (‖L) において s は $s \parallel t$ の部分項）．よって (4) のような導出は，左側の項がだんだん「太っていく」形で行われる．

6　SOS の形式検証への応用

前節で，プロセス代数の項——すなわち，並列システムを表現する単純なプログラム——に対して (3) のような意味を定義する手法 SOS を紹介した．数学的・論理学的にはいかにもまっとうだが，計算機科学（特に本稿冒頭でふれた**形式検証**）ではこれをどのように**使う**のだろうか？

まず，現在の文脈で形式検証とは次の問題である．

- **入力** プロセス代数の項 t と，みたしてほしい性質 P．P の例は「$\mathtt{a}$ が現れたら，その後いつかかならず $\mathtt{b}$ が現れる」．
- **出力** 項 t の表すシステム（すなわち遷移系 $[\![t]\!]$）が P をみたすか否か（Yes/No）．No の場合はできれば反例．

この問題は（主に経済的理由から）できれば自動化して解きたい：すなわち，形式検証問題を解くアルゴリズムを実装したいのである．

ここで図 1 によると，項 $!s$ は再帰呼び出しを行うゆえ，(2) のプログラムのように無限状態の遷移系をひきおこすことが多い．この場合には (3) のよ

うな意味を具体的に書きくだすのは不可能である（その前に我々の有限の寿命が尽きる）.

そこで多くの形式検証アルゴリズムは，項 t を遷移系 $[\![t]\!]$ に完全に展開することはせず，何らかの記号操作によって（記号列として与えられる）項 t と性質 P を変換していき Yes/No の出力に至る．そしてこのようなアルゴリズム——振り返ってみると文字列 t, P を機械的に変換しているだけである——が実際に形式検証問題を正しく解いているのかという問題は決して自明ではない．つまり「検証プログラム自体の（メタな）検証」が必要なのである．

この問題を（$[\![t]\!]$ を定義することで）定式化し，またその証明の手がかりを与えるのが，前節のような数学的意味論である．言いかえると，形式検証アルゴリズムの**メタ理論**（meta theory）を与えるのが数学的意味論なのである．

また逆に，数学的意味論をスジよく発展させることで生まれる形式検証手法もある．たとえばプロセス代数の**要素還元性**（compositionality）とよばれる性質は「小さなシステムを部品として組み合わせて作った大きなシステムの性質は，部品それぞれの性質からわかる」ことをいい，次のように定式化される： 各演算子 σ について

$$[\![s_i]\!] \simeq [\![t_i]\!],\ \forall i \in [1,n] \quad \Longrightarrow \quad [\![\sigma(s_1,\ldots,s_n)]\!] \simeq [\![\sigma(t_1,\ldots,t_n)]\!]. \quad (6)$$

ここで $\simeq$ は遷移系の間の（何らかの意味での）等価性を表す[8]．たとえば「s_1 さんと t_1 さんの個人としての働きが同じで，s_2 と t_2 も同じであれば，s_1, s_2 二人のチームの働きは t_1, t_2 二人のチームの働きと同じ」という感じだが，現実にはむずかしそうだ： 実は s_1 と s_2 は犬猿の仲であって，それぞれ個人として働いているうちには露見しなかっただけかもしれない．このような**予見しない内部干渉がない**という性質が要素還元性であり，複雑なシステムの形式検証をボトムアップに（部分部分から積み上げる形で）行うためには必要不可欠な性質である．

要素還元性は上記の（代数学に基づいた） SOS の格好のターゲットである： 条件 (6) は，意味を与える写像 $[\![-]\!]$ が準同型写像であることに他ならない！[9] 現在では要素還元性を保証するルールの形——5 節に述べたフォーマット——が多く研究され，たとえば図 1 のルールが要素還元性をみたすことは，あるフォーマットからすぐにわかる．

7　おわりに

計算機科学（特に意味論）と代数学における**有限と無限の対比**，また**構文論と意味論の分離**についてインフォーマルに説明したのち，Plotkin の構造

[8] たとえば前頁の脚注で述べた双模倣性.

[9] 正確には：遷移系の $\simeq$ 同値類の上に演算子 σ の意味 $[\![\sigma]\!]$ が well-defined であり，かつ $[\![-]\!]$ が Σ 項の集合自身がなす代数からの準同型写像になっていること.

的操作意味論 SOS をかけ足で紹介した．SOS の入門は Aceto らによるもの [1] があるが，SOS の**圏論的代数・余代数**を用いた一般的な記述（Turi と Plotkin による）もエキサイティングでありオススメである（Klin による入門記事 [2] がある）．論理学における構文論と意味論の分離については，論理学の高級教科書である Shoenfield[3] が鬼軍曹のように鍛えあげてくれる．

本稿は計算機科学における意味論のわずかな一側面を語ったにすぎない．意味論の代表的な教科書は Winskel[4] であるが，プログラム意味論と幾何学的不変量のつながり [6] や，代数仕様を記述するための圏論的道具である**モナド**（monad）の計算機科学での使われ方 [5] なども読者の興味のあるところであろう．

参考文献

[1] L. Aceto, W. Fokkink and C. Verhoef. Structural operational semantics. In J. Bergstra, A. Ponse and S. Smolka, editors, **Handbook of Process Algebra**, pp. 197–292. Elsevier, 2001.

[2] B. Klin. Bialgebras for structural operational semantics: An introduction. **Theor. Comput. Sci.**, 412(38):5043–5069, 2011.

[3] J.R. Shoenfield. **Mathematical Logic**. Addison-Wesley, 1967.

[4] G. Winskel. **The Formal Semantics of Programming Languages**. MIT Press, 1993.

[5] 勝股 審也. モナドと計算効果. 圏論の歩き方 (第 5 回)，数学セミナー 2011 年 12 月号.

[6] 長谷川 真人. プログラム意味論と圏論—計算の「不変量」を圏論で捉える. 圏論の歩き方 (第 4 回)，数学セミナー 2011 年 11 月号.