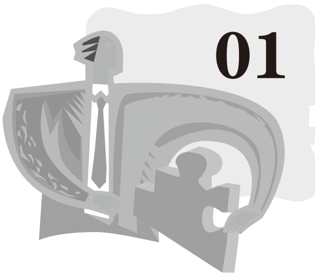
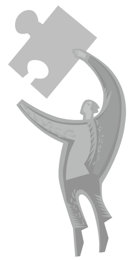


01 プログラミング言語の数学



住井英二郎（東北大学）

プログラミング「言語」の数学とは

数学がプログラミングに有用である，という考えに反対する人はあまりいないだろう．無論，高度な数学を必要としないプログラムも少なくはないが，各種のアルゴリズムや，画像処理を含むコンピュータグラフィクス，機械学習を始めとする人工知能，あるいは暗号等，中学レベルから大学院レベルまで，実にさまざまな数学が活躍している．

では，プログラミング「言語」に数学は有用だろうか．プログラミング言語というと，本誌の読者の方の多くはC，C++，Java，JavaScript，PHP，Python，Ruby 等を思い浮かべると思う☆1．いわゆる関数型言語（Haskell，ML 等）や論理型言語（Prolog 等）はまだしも，前述のような言語に「数学」が関係するようにはあまり思われないかもしれない．

しかし実際には，複雑化する一方の計算機ソフトウェアないし情報システム一般に対し，それらに含まれるプログラムおよびプログラムを記述する言語自体も複雑化し，もはや数学的・論理学的理論に基づかない「人間の勘」だけでは手に負えなくなっている．ネットワークや組込みシステム等の安全性に直にかかわるプログラムに致命的なミスが見つかったり☆2，プログラミング「言語」自体の仕様に想定外の問題があったり☆3，といった事件は今や日常茶飯事だ．

プログラムに対する数理論理学的検証手法としては型システム[1]，定理証明器☆4，ソフトウェアモデル検査[2]等がある．プログラムはプログラミング言語により記述されるので，これらのプログラム検証手法は当然ながらプログラミング言語自体の数理論理学的理論を基礎に成り立っている．本稿ではそのような理論のうち，特に基本的な部分を，中学〜大学1年程度の数学のみを用いて，できるだけ平易に（ただし単なる表面的な「おはなし」だけでなく実質的詳細にも踏み込んで）紹介したい．

帰納的定義と構造的帰納法

プログラムは無限に存在する．たとえば，仮に整数の引き算しか書けない，非常に単純な言語を考えてみても，「3−7」「42−(3−7)」「42−(3−7))−10」等々，いくらでも複雑な式を作ることができる．

しかし，プログラムやプログラミング言語に関する検証を計算機上で行うにせよ紙の上で行うにせよ，計算機のメモリや紙のスペースは有限だ．そのような有限の記述で，無限に存在するプログラムに関する定義や証明をどうして行えるだろうか？

ここで早速だが「数学」の助けを借りよう．数学で「無限に存在する」対象として最も基本的なものは「自然数」である．無限に存在する自然数に関する議論──たとえばすべての自然数 n について，$1^3+2^3+3^3+\ldots+n^3=n^2(n+1)^2/4$ が成り立つことの証明──には，高校で習う「数学的帰納法」が有効だ．つまり，まず「$n=1$ の場合」を考え☆5，次に「任意の自然数 k について，$n=k$ の場合を仮定して $n=k+1$ の場合」を考えれば，$n=1, 2, 3, \ldots$ とドミノ倒しのよ

☆1 TIOBE Index（http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html）という，ある種の「プログラミング言語の人気ランキング」から適当に抜粋した．

☆2 あまりにも多いため，例を挙げるとキリがないが，Heartbleed 脆弱性（http://heartbleed.com/）等．

☆3 これももはや珍しくないが，たとえば当時の JVM に対し Saraswat が発見した問題（“Java is not type-safe”）が有名．

☆4 Coq（http://coq.inria.fr/）等．

☆5 現代の日本の高校では自然数は1から始まる！

うに「すべての自然数 n」を考えたことになる．

このような数学的帰納法の考え方を応用して，引き算プログラムの集合 S を次のような条件により定義してみよう．

1. 整数定数式 ..., −3, −2, −1, 0, 1, 2, 3, ... は S に属する．
2. S に属する任意の式 E_1, E_2 について，式 E_1-E_2 は S に属する．

上の 1 番の条件により，3 や 7 は S に属することが分かる．すると 2 番の条件により「3−7」という式も S に属することが分かる（「3−7」という式自体が S に属するのであって，式を計算した値 −4 の話ではないことに注意されたい）．さらに 2 番の条件を繰り返し用いることにより，「42−(3−7)」「(42−(3−7))−10」等々も S に属することが分かる．ただしここでは簡単のため，括弧は人間が適切に補うものとする．別の言い方をすると，文字列ではなく最初から構文木を考えることにする．このように木構造のみを考えた構文を抽象構文，逆に括弧等も考慮した文字列としての構文を具象構文という．

果たして，これで無限集合 S を有限の条件で定義できた…のだろうか？　例が簡単なので，「何を大げさな」とも思われるかもしれない．ところが，よく考えると上の条件 1, 2 は，S が満たすべき「条件」を述べているだけであって，S の「定義」の体を成していない．実際，上述の条件 1, 2 を満たす S は無数に存在する！　たとえば整数だけでなく浮動小数点数とそれらに対する引き算も追加した集合 S' は，条件 1, 2 を満たす．もちろん，今は浮動小数点数など「想定外」なので，数学的定義や論理的仕様として，これではまずい．

そこで，浮動小数点数のような余計な「ゴミ」が混ざらないよう，集合 S を「条件 1, 2 を満たすすべての集合の共通部分」と定義する．数式で書くと次のとおりだ．

$$S = \cap\{S' \mid ..., -3, -2, -1, 0, 1, 2, 3, ... \in S' \wedge \forall E_1, E_2 \in S'.\ E_1-E_2 \in S'\}$$

これはれっきとした「S の定義」になっている．S 自体は = の右辺のどこにも現れていないことに注意されたい．

このように，ある種の条件を満たす集合の共通部分[☆6]としての集合の定義を「帰納的定義」と言う．帰納的に定義された集合に対しては，数学的帰納法を一般化した「構造的帰納法」という証明手法が適用できる[☆7]．これにより，たとえば上述の S に属する式の値は確かに整数になる（後述）等，無限に存在するプログラムに関する一般的性質が，有限の議論で証明できるようになる．

帰納的定義や構造的帰納法は，以下の「操作的意味論」「型システム」，あるいは定理証明器を用いた検証・定式化でも必須となる．プログラミング言語理論で最も基本的な「数学」と言ってもよいだろう．

表示的意味論と操作的意味論

先の集合 S は「引き算言語」の（抽象）構文のみを定義していた．では，S の要素すなわちプログラムの「意味」はどう定義すべきだろうか？

この言語の場合, 最も素直な意味の定義（意味論）は，以下のように各プログラムに対し「値」を定めることだろう．

- 整数定数式 i の値は整数 i そのものとする．
- 引き算式 E_1-E_2 の値は，E_1 の値を i_1，E_2 の値を i_2 として，（普通の数学の意味で）i_1 から i_2 を引いた結果の整数とする．

この定義がプログラムから整数への「関数」を与えている，すなわち S の任意の要素 E に対して一意な整数を定めていることは，E に関する構造的帰納法により，以下のとおり証明できる．

- E が整数定数式 i の場合，E の値は i そのものなので，一意な整数である．
- E が引き算式 E_1-E_2 の場合，帰納法の仮定より，E_1 と E_2 の値はそれぞれ一意な整数 i_1, i_2 なので，i_1 から i_2 を引いた結果すなわち E の値も一意な整数である．

☆6　条件を満たす「最小の集合」と定義しても同値である．

☆7　というか，そもそも自然数の集合 $\mathcal{N}$ も「0 は $\mathcal{N}$ に属する」「$\mathcal{N}$ に属する任意の n について，$n+1$ は $\mathcal{N}$ に属する」と帰納的に定義される．

値の一意性はこの言語では当たり前すぎて，かえって禅問答のように感じられるかもしれないが，仮に非決定性のある言語だったら「ある1つの式Eの値が3にも7にもなる」といった可能性もあるので，証明が必要な性質である．

さて，このようなプログラムから何らかの数学的な値（上の例では整数）への関数を定義する意味論（表示的意味論という）は，言語が高度になるにつれて，より高度な数学を必要とする．たとえば停止しない可能性のある再帰関数やwhileループ等があると，プログラムの意味論を無限列の「極限」として与えるため，完備半順序（CPO）等の位相論が必要になる（ただしそれでもある意味で「不完全」な意味論にしかならない場合があることが知られている）．「並列性」はさらに数学的扱いが難しい．

このような困難があるため，再帰やループないし並列性等のある言語における現実的な検証では，(それはそれで面白いのだが）高度な数学を必要とする表示的意味論よりも，より平易な「操作的意味論」を用いることが多い．

操作的意味論は，簡単にいえばプログラムの意味を状態遷移系ないし書き換え系で表す意味論である[☆8]．たとえば上述の引き算言語で(42−(3−7))−10というプログラムの意味は

$$\begin{aligned}(42-(3-7))-10 &\to (42-(-4))-10\\ &\to 46-10\\ &\to 36\end{aligned}$$

のような遷移（「簡約」と言う）で与えることができる．より一般的に，S上の二項関係$\to$を以下のように帰納的に定義することができる．

1. 任意の整数i, jについて，iからjを引いた結果がkならば$i-j\to k$が成り立つ．
2. Sの任意の要素E_1, E_1', E_2について，$E_1\to E_1'$ならば$E_1-E_2\to E_1'-E_2$が成り立つ．
3. 任意の整数iと，Sの任意の要素E_2, E_2'について，$E_2\to E_2'$ならば$i-E_2\to i-E_2'$が成り立つ．

たとえば先の簡約のうち，3ステップ目の46−10→36は上の1番の条件より成り立つ．同じく1番の条件より42−(−4)→46も成り立つので，それと2番の条件より，2ステップ目の(42−(−4))−10→46−10も成り立つ．また，1番の条件より3−7→−4，それと3番の条件より42−(3−7)→42−(−4)，さらにそれと2番の条件より，1ステップ目の(42−(3−7))−10→(42−(−4))−10が成り立つ．

(42−(3−7))−10→36のように途中を省略した簡約は，→の定義に従えば成り立たないことに注意してほしい．複数回の簡約を表したいときは，$(42-(3-7))-10\to^* 36$のように，→の反射的推移的閉包$\to^*$（0個以上の→を合成した関係）を用いればよい．

3番の条件で，−の左側は任意の式E_1ではなく整数iに限っていることにも注意されたい．この制限により，たとえば(1−2)−(4−3)→(1−2)−1のように−の右側を左側より先に簡約することはできない（ように定義している）．このように「左から右への」計算順序を強制することは，簡約の各ステップを完全に決定的とし，結果の一意性などを自明にするメリットがある一方，並列化や最適化の妨げになる可能性もある．

型システム

引き算言語Sに大小判定$E_1 < E_2$と論理値`true`, `false`および条件分岐`if` E_1 `then` E_2 `else` E_3を追加した言語Tを考えてみよう．追加した部分に対する操作的意味論は以下のように定義できる（「任意の…について」の部分は省略する）．

- iがjより小さければ$i<j\to$ `true`，そうでなければ$i<j\to$ `false`
- `if true then` E_2 `else` $E_3\to E_2$ かつ `if false then` E_2 `else` $E_3\to E_3$
- $E\to E'$ならば，$E<E_2\to E'<E_2$かつ$i<E\to i<E'$かつ`if` E `then` E_2 `else` $E_3\to$ `if` E' `then` E_2 `else` E_3

☆8 ここでは正確には一気に結果を得る「大ステップ」の操作的意味論ではなく，1つ1つの状態遷移を追う「小ステップ」の操作的意味論を説明する．

たとえば `if` $1<2$ `then` 3 `else` $4-5 \rightarrow$ `if false then` 3 `else` $4-5 \rightarrow 4-5 \rightarrow -1$ という簡約が成り立つ．

しかし，整数と引き算しかない言語 S と異なり，この言語 T ではたとえば `if` 42 `then` 3 `else` 7 のような，簡約ができない（結果の値が得られない）プログラムも書けてしまう．このように，まだ結果の値が得られていないにもかかわらず，それ以上の簡約ができない状態を，行き詰まり（stuck）状態という．行き詰まり状態は，直観的には実行時エラーを表しているものとみなすことができる．

このような実行時エラーを防ぐには静的型付けが有効だ．具体的には，2 つの型 `int` と `bool` および以下の帰納的定義（型付け規則）を考える．

1. 整数定数式 i は型 `int` を持つ．
2. 論理値 `true`, `false` は型 `bool` を持つ．
3. 式 E_1, E_2 が型 `int` を持つならば，式 E_1-E_2 は型 `int` を持ち，式 $E_1<E_2$ は型 `bool` を持つ．
4. 式 E_1 が型 `bool` を持ち，かつ式 E_2, E_3 が何らかの型 T を持つならば，式 `if` E_1 `then` E_2 `else` E_3 も同じ型 T を持つ．

たとえば `if` $1<2$ `then` 3 `else` $4-5$ という式は型 `int` を持つ一方，`if` 42 `then` 3 `else` 7 という行き詰まり状態の式はいかなる型も持たない（ここでも「余分なゴミが混ざっていない」という帰納的定義の性質が重要となる）．

より一般的に「何らかの型を持つ式は，簡約しても決して行き詰まり状態にならない」という定理（型安全性）が成り立つ．証明は以下の 2 つの補題による．

- 何らかの型を持つ式は，簡約しても同じ型を持つ．これを主部簡約（subject reduction）と言う．
- 何らかの型を持つ式は，直ちに行き詰まり状態にはない．すなわち，すでに値（整数定数または論理値）であるか，あるいは少なくとも 1 回は簡約できる．

これらの補題自体の証明は，型付け規則に関する構造的帰納法による．

主部簡約「だけ」では，ほとんど意味がないことに注意されたい．たとえば任意のプログラムに任意の型を与えてしまう（明らかに誤った）型付け規則の下でも，主部簡約だけであれば自明に成り立つ．なお，逆に「すべてのプログラムが，いかなる型も持たない」型システムは，役に立たないが，安全ではある．

型システムは，`int` と `bool` のような単純な区別だけでなく，「抽象型」により実装を隠蔽したり，「0 でない整数」や「必ず `true` になる式」といった「詳細型」によりアサーションの成功を静的に保証したり等，より高度なプログラム検証にも応用可能である [3]．

参考文献

1) Pierce, B. C.：型システム入門―プログラミング言語と型の理論，オーム社 (2013).
2) Jhala, R. and Majumdar, R. : Software Model Checking, *ACM Computing Surveys*, 41(4):21:1-21:54 (2009).
3) Pierce, B. C. editor. : *Advanced Topics in Types and Programming Languages*, MIT Press (2005).

(2015 年 3 月 1 日受付)

住井英二郎（正会員）■ sumii@ecei.tohoku.ac.jp

1998 年東京大学理学部情報科学科卒業，2004 年東京大学博士（情報理工学），2014 年東北大学大学院情報科学研究科教授．日本学術会議連携会員（若手アカデミー会員），Global Young Academy メンバー．