

Type Systems and Programming Language, 2-3.4 章

坂本 崇裕

2001 年 4 月 16 日

2 数学的準備

本論に入る前に, 共通の表記法や数学的事項について述べる. 必要に応じて読み返すようにしてもらえればいい.

2.1 集合, 関係, 関数

定義 明示的に要素を列挙した集合 ($\{\dots\}$), 他の集合から生成するルールを記述した集合 ($\{x \in S \mid \dots\}$), 空集合 (\emptyset), 差集合 ($S \setminus T$), 集合 S の要素の数 ($|S|$), 集合 S の全部分集合の集合 ($P(S)$).

定義 $\{0, 1, 2, 3, 4, \dots\}$ の自然数の集合は \mathbf{N} . 集合の各要素が自然数と対応付けられるならば, その集合は可算集合という.

定義 集合 (S_1, \dots, S_n) 間の n 項関係は集合 $R \subseteq S_1 \times S_2 \times \dots \times S_n$ で表される. 要素 $s_1 \in S_1, \dots, s_n \in S_n$ は (s_1, \dots, s_n) が R の要素 であるときに R に関係付けられているという.

定義 集合 S における 1 項関係は S における述語と呼ぶ. P に $s \in S$ なる要素が与えられたとき $s \in P$ ならば真などという. $s \in P$ の代わりに $P(s)$ と記述して, P を S の要素から真偽値への写像関数と見ることがある.

定義 2 つの集合 S, T 間の関係 R のことを *binary relation* (2 項関係) と呼ぶ. $(s, t) \in R$ の代わりに $s R t$ と記述することがある, また S, T が同じ集合 U であるときに R は U における 2 項関係と呼ぶ.

定義 読みやすさのために 3 項以上の関係は *mixfix* で記述することがある. 各要素をシンボルで区切った記述法をする. (例. $\Gamma \vdash s : T$)

定義 関係 $R \subseteq S \times T$ の domain $dom(R)$ とは $\{s \in S \mid \exists t. (s, t) \in R\}$ なる集合であり, 同様に codomain もしくは range $range(R)$ は $\{t \in T \mid \exists s. (s, t) \in R\}$ なる集合である.

定義 関係 $R \subseteq S \times T$ が任意の $(s, t_1) \in R$ かつ $(s, t_2) \in R$ という関係において $t_1 = t_2$ であるとき R は S から T への部分関数 という. さらに $dom(R) = S$ であるときは全域関数 (または単に関数) という.

定義 部分関数 $R \subseteq S \times T$ は引数 $s \in S (s \in dom(R))$ において定義されているといい, そうでないときは定義されていないという. $f(x) \uparrow$ または $f(x) = \uparrow$ は “ f は x において定義されていない”, $f(x) \downarrow$ で “ f は x において定義されている” を表す.

定義 R を集合 S における関係, P を S における述語とする. $s R s'$ かつ $P(s)$ であるすべての場合において $P(s')$ であるとき, P は R に *preserve* されているという.

2.2 順序集合

定義 集合 S における関係 R が, 反射的: $\forall s \in S. s R s$, 対称的: $s R t$ なる任意の S の要素 (s, t) で $t R s$, 推移的: $s R t$ かつ $t R u$ なる任意の S の要素 (s, t, u) で $s R u$. 非対称的: $s R t$ かつ $t R s$ なる任意の S の要素 (s, t) で $s = t$.

定義 集合 S における関係 R が反射的かつ推移的かつ非対称的であるとき S における半順序と呼ぶ. 半順序は \leq もしくは \sqsubseteq で表す. さらに, 任意の S の要素 s, t において半順序が定義されている場合は全順序と呼ばれる.

定義 集合 S, S 上での半順序 \leq, S の要素 s, t, j, m において j が以下の性質を満たすとき j を s, t の *join* (または least upper bound) と呼ぶ.

1. $s \leq j$ かつ $t \leq j$ かつ
2. $s \leq k$ かつ $t \leq k$ なる任意の $k \in S$ で $j \leq k$.

同様に m が以下の性質を満たすとき m を s, t の *meet* (または greatest lower bound) と呼ぶ.

1. $m \leq s$ かつ $m \leq t$ かつ
2. $n \leq s$ かつ $n \leq t$ なる任意の $n \in S$ で $n \leq m$.

定義 反射的かつ推移的かつ対称的な関係を同値関係と呼ぶ.

定義 R を集合 S における関係とすると, 反射閉包: R を含む最小の反射的關係, 推移閉包: R を含む最小の推移的關係 R^+ , 反射推移閉包: R を含む最小の反射的かつ推移的關係 R^* ,

2.3 帰納法

定義 集合 S における半順序 \leq を考える. \leq における減少列とは s_1, s_2, \dots という S の要素の数列のうち $\forall i. s_{i+1} < s_i$ を満たすものである.

定義 集合 S における半順序 \leq を考える. 無限長の減少列を持たないとき, \leq は *well founded* であるという. たとえば, 自然数における通常の順序付けは *well founded* であるが, 整数に対しては *well founded* ではない.

公理 PRINCIPLE OF WELL-FOUNDED INDUCTION

集合 S において *well-founded* な順序 \leq と 述語 P があるとき,

If, for each $s \in S$,
 assuming $P(r)$ (for all $r < s$)
 we can show $P(s)$,
then
 $P(s)$ is true for all s .

系 PRINCIPLE OF COMPLETE INDUCTION ON NATURAL NUMBERS

自然数において P を述語とすると,

If, for each natural number n
 assuming $P(i)$ (for all $i < n$)
 we can show $P(n)$,
then
 $P(n)$ is true for all n .

証明: 自然数における通常の順序は *well-founded* であるため.

定義 自然数のペアにおける辞書的順序を以下のように定義する:

$$(m, n) \leq (m', n') \text{ iff } (m < m') \text{ もしくは } (m = m' \text{ かつ } n \leq n')$$

系 PRINCIPLE OF LEXICOGRAPHIC INDUCTION

P を自然数のペアにおける述語とする. もし任意の (m, n) に関して $P(m, n)$ が $\forall(m', n') < (m, n). P(m', n')$ から導けるならば, $P(m, n)$ は任意の (m, n) で真である.

2.4 数列

定義 順序数列は要素をカンマ', 'で区切って並べる. カンマは `cons` の意味にも `append` の意味にも用いる. $1, 2, \dots, n$ の数列は $1..n$ と短縮表記する. 数列 a の長さは $|a|$ と表し, 空数列は \cdot と表す.

3 Untyped Arithmetic Expressions

型システムとその特質について厳密に語るために, プログラム言語の基本的観点について形式的に取り扱わねばならない. 特に, プログラム言語の文法や操作を表したり証明したりする明確かつ正確かつ数学的に取り扱いやすい道具が必要となる.

この章と次の章では, 数と真偽値だけのとても小さな言語向けにその道具を開発する.

3.1 Introduction

この章で取り上げる言語はとても小規模であり, その文法は以下のようにまとめられる.

<code>t ::=</code>	<i>terms:</i>
<code> true</code>	<i>constant true</i>
<code> false</code>	<i>constant false</i>
<code> if t then t else t</code>	<i>conditional</i>
<code> 0</code>	<i>constant zero</i>
<code> succ t</code>	<i>successor</i>
<code> pred t</code>	<i>predecessor</i>
<code> iszero t</code>	<i>zero test</i>

ここで使用した文法の表現形式は standard BNF (Aho, Sethi, and Ullman, 1986) という形式に近い. 最初の行 (`t ::=`) は `term` の集合を定義していることを示し, 以下で `t` というシンボルを任意の `term` に置き換わるものとして使用する. 以下の各行ではそれぞれひとつの syntactic な `term` の形式を示している. 右側のイタリックの部分は単なるコメントである.

右側に出てくる `t` をメタ変数と呼ぶ. 以下で `t` 付近の文字をその言語の `term` を表すメタ変数と扱う.

```
  if false then true else false;
> false

  0;
> 0

  pred (succ 0);
> 0

  iszero (pred (succ 0));
> true

  iszero (succ (succ 0));
> false
```

括弧に囲む表現は単に読みやすさのためである. さらに簡潔のために, 本来は数は 0 に対する `succ` の適用回数で表されるべきところをこの評価器ではアラビア数字でも表示することにする.

表記例

```
succ (succ (succ 0));
> 3

succ 3;
> 4
```

評価の結果表示されるのは真偽値か数字のみである。これらの term のことを値 (value) と呼び、term の評価順序を形式化する際に重要な意味を持つ。

3.2 Syntax

この言語の syntax を表現する等価な方法はいくらか存在する。前節で示したのもそのひとつであるが、これは次に示す帰納的定義の簡潔な記述である。

TERMS, INDUCTIVELY

この言語の term の集合は以下の条件を満たす最小の集合 \mathcal{T} である

1. $\{\text{true}, \text{false}, 0\} \in \mathcal{T}$;
2. if $t_1 \in \mathcal{T}$, then $\{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1\} \in \mathcal{T}$
3. if $t_1 \in \mathcal{T}, t_2 \in \mathcal{T}, t_3 \in \mathcal{T}$, then $\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in \mathcal{T}$

1 行目は 3 つの単純な表現が \mathcal{T} に含まれること、2,3 行目は複合式が \mathcal{T} に含まれるかどうかを判定するルールを示しており、最後に“最小の”集合というのはここから生成される以外の term を持たないことを意味する。

あとで処理することになるが、ここで“意味をなさない”term について問題となる。たとえば $\text{if } 0 \text{ then } 0 \text{ else } 0$ とか succ false のようなものである。このようなものを阻止する仕組みは上記定義にはないが、8 章までこの問題は保留する。

同じ帰納的定義の別の表記として、分数形式のいわゆる“natural deduction style”という表現形式がある。

TERMS, BY INFERENCE RULES

$$\begin{array}{c} \text{true} \in \mathcal{T} \quad \text{false} \in \mathcal{T} \quad 0 \in \mathcal{T} \\ \\ \frac{t_1 \in \mathcal{T}}{\text{succ } t_1 \in \mathcal{T}} \quad \frac{t_1 \in \mathcal{T}}{\text{pred } t_1 \in \mathcal{T}} \quad \frac{t_1 \in \mathcal{T}}{\text{iszero } t_1 \in \mathcal{T}} \\ \\ \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in \mathcal{T}} \end{array}$$

最初の 3 つのルール (仮定節がないので正確には公理) は前節の 1 行目を言い換えたもので、次の 4 つのルールは前節の 2,3 行目を言い換えたものである。それぞれのルールは“線の上の仮定を満たすならば、線の下結論を導かれる”という意味である。 \mathcal{T} がルールを満たす最小の集合ということは暗黙事項となっている。

最後に、同じ term の集合を定義する少し異なった方法である。 \mathcal{T} の要素を生成する明示的な方法を与える具体的なものである。

TERMS, CONCRETELY

すべての自然数 i に対し 集合 S_i を以下のように定める:

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= \{\text{true}, \text{false}, 0\} \\ &\cup \{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 \mid t_1 \in S_i\} \\ &\cup \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in S_i\} \end{aligned}$$

そして

$$S = \bigcup_i S_i.$$

S_i の要素は累積的 ($S_i \subseteq S_{i+1}$) に増えていく (Exercise). こうして異なる方法で定義された \mathcal{T} と S は同じものである.

命題 $\mathcal{T} = S$

証明: \mathcal{T} はいくつかの条件を満たす最小の集合と定義されているので, (a) S がそれらの条件を満たすこと, (b) それらの条件を満たすすべての集合が S を部分集合として持つこと, を示せばよい.

(a). 1つめの条件: S_1 が $\{\text{true}, \text{false}, 0\}$ を含むことから S も含んでいる. 2つめの条件: $t_1 \in S$ であるとき, 定義からある i があって $t_1 \in S_i$. そして S_{i+1} の定義から $\text{succ } t_1 \in S_{i+1}$ すなわち $\text{succ } t_1 \in S$. 同様にして $\text{pred } t_1 \in S$, $\text{iszero } t_1 \in S$. 3つめの条件: S_i の要素は単調増加することから, $t_1, t_2, t_3 \in S$ であるとき $t_1, t_2, t_3 \in S_i$ なる i が存在し, 以下同様の議論.

(b). 3つの条件をみたす任意の集合を S' としたとき, すべての i で $S_i \subseteq S'$ となることを言うことで $S \subseteq S'$ を示す.

すべての $j < i$ で $S_j \subseteq S'$ を仮定し, $S_i \subseteq S'$ を示す. S_i の定義は2節からなるので, それぞれについて考えなければならない.

- $i = 0$ のとき $S_i = \emptyset$. $\emptyset \subseteq S'$ は自明.
- $i = j + 1, (j : \text{自然数})$ のとき, t を S_{j+1} の要素とすると, S_{j+1} の定義から t は3つの節のどれかから導き出されたものである.
 1. t が定数. $t \in S'$ (1つめの条件より)
 2. t が $\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1$ の形であり, $t_1 \in S_j$. このとき仮定から $t_1 \in S'$, さらに2つめの条件から $t \in S'$.
 3. t が $\text{if } t_1 \text{ then } t_2, \text{ else } t_3$ の形であり, $t_1, t_2, t_3 \in S_j$. 上と同様の議論で $t \in S'$.

以上から $\forall i. S_i \subseteq S'$. それと S の定義から $S \subseteq S'$.

3.3 Induction on Terms

$\mathcal{T} = S$ であることから, $t \in \mathcal{T}$ であるならば以下が成り立つ:

1. t は定数, もしくは
2. t はより短い term t_1 で $\text{succ } t_1$ または $\text{pred } t_1$ または $\text{iszero } t_1$ と表せる, もしくは
3. t はより短い term t_1, t_2, t_3 で $\text{if } t_1 \text{ then } t_2 \text{ else } t_3$ と表せる.

この事実には2通りの使い道がある: (1) terms に関する関数の帰納的定義を与える, (2) terms に関する特性に帰納的証明を与える.

定義 Term t に現れる定数の集合 $Consts(t)$ は以下のように定義できる:

$$\begin{aligned}
Consts(\mathbf{true}) &= \{\mathbf{true}\} \\
Consts(\mathbf{false}) &= \{\mathbf{false}\} \\
Consts(0) &= \{0\} \\
Consts(\mathbf{succ } t_1) &= Consts(t_1) \\
Consts(\mathbf{pred } t_1) &= Consts(t_1) \\
Consts(\mathbf{iszero } t_1) &= Consts(t_1) \\
Consts(\mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3) &= Consts(t_1) \cup Consts(t_2) \cup Consts(t_3)
\end{aligned}$$

定義 Term t のサイズ $size(t)$ は以下のように定義できる:

$$\begin{aligned}
size(\mathbf{true}) &= 1 \\
size(\mathbf{false}) &= 1 \\
size(0) &= 1 \\
size(\mathbf{succ } t_1) &= size(t_1) + 1 \\
size(\mathbf{pred } t_1) &= size(t_1) + 1 \\
size(\mathbf{iszero } t_1) &= size(t_1) + 1 \\
size(\mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3) &= size(t_1) + size(t_2) + size(t_3) + 1
\end{aligned}$$

これは abstract syntax tree のノード数になっている。同様に、

定義 Term t の AST におけるノードの深さ $depth(t)$ は以下のように定義できる:

$$\begin{aligned}
depth(\mathbf{true}) &= 1 \\
depth(\mathbf{false}) &= 1 \\
depth(0) &= 1 \\
depth(\mathbf{succ } t_1) &= depth(t_1) + 1 \\
depth(\mathbf{pred } t_1) &= depth(t_1) + 1 \\
depth(\mathbf{iszero } t_1) &= depth(t_1) + 1 \\
depth(\mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3) &= \max(depth(t_1), depth(t_2), depth(t_3)) + 1
\end{aligned}$$

$depth(t)$ は $t \in S_i$ となる最小の i と同じである。

次に term サイズとその中の定数の数に関する補題の帰納的証明を示す。この補題のプロパティ自体はほぼ自明であるが、その証明の仕方がポイントである。

補題 Term t 中の定数の種類の数は t のサイズを超えない。 (i.e., $|Consts(t)| \leq size(t)$.)

証明: t の $depth$ に関する帰納法による。 t より小さい任意の term において題意の性質が満たされると仮定し、 t においてその性質を証明する。考えなければならないケースは 3 つ。

Case: t が定数

$$|Consts(t)| = |t| = 1 = size(t)$$

Case: $t = \mathbf{succ } t_1$ または $\mathbf{pred } t_1$ または $\mathbf{iszero } t_1$

$$\text{仮定より } |Consts(t_1)| \leq size(t_1). \text{ 従つて } |Consts(t)| = |Consts(t_1)| \leq size(t_1) < size(t).$$

Case: $t = \mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3$

$$\text{仮定より } |Consts(t_1)| \leq size(t_1), |Consts(t_2)| \leq size(t_2), |Consts(t_3)| \leq size(t_3), \text{ 従つて}$$

$$\begin{aligned}
|Consts(t)| &= |Consts(t_1) \cup Consts(t_2) \cup Consts(t_3)| \\
&\leq |Consts(t_1)| + |Consts(t_2)| + |Consts(t_3)| \\
&\leq size(t_1) + size(t_2) + size(t_3) \\
&< size(t)
\end{aligned}$$

この証明の形を、他の似ている 2 種類の関数も含めてまとめてみると以下のようなになる。

定理 PRINCIPLES OF INDUCTION ON TERMS P を terms における述語とすると、

Induction on depth:

If, for each term s ,
 assuming $P(r)$ (for all r s.t. $depth(r) < depth(s)$)
 we can show $P(s)$,
then
 $P(s)$ is true for all s .

Induction on size: If, for each term s ,

 assuming $P(r)$ (for all r s.t. $size(r) < size(s)$)
 we can show $P(s)$,
then
 $P(s)$ is true for all s .

Structural induction: If, for each term s ,

 assuming $P(r)$ (for all immediate subterms r of s)
 we can show $P(s)$,
then
 $P(s)$ is true for all s .

Term の $depth$ と $size$ に関する帰納法は COMPLETE INDUCTION ON NATURAL NUMBERS (2.3) に似ている。構造的帰納法はもっと馴染み深い数学的帰納法 ($P(n+1)$ を証明するのに $P(n)$ しか使わない) に似ている。

単純な証明では上の 3 種の証明法のどれを使っても大差ない。しかし遠回りを避けるため、できる限り構造的帰納法を用いるのが一般的である。

どの帰納法を用いても証明はほとんど同じ構造になる。各帰納のステップにおいて、与えられた term t と述語 P に対し $P(t)$ を t の subterm において P が真であると仮定して証明する。これを t のとりうる形に場合わけして行うわけである。そのなかでの違いは、各場合分けでの個々での議論の部分だけである。一般的には冗長な部分を省略して以下のように書くのが普通である。

Proof: By induction on t .

Case: $t = \text{true}$

 ...show $P(\text{true})$...

Case: $t = \text{false}$

 ...show $P(\text{false})$...

Case: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$

 ...show $P(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)$, using $P(t_1)$, $P(t_2)$, and $P(t_3)$...

(And similarly for the other syntactic forms.)

“By induction on t .” という文句は必須項目である。

3.4 Semantic Styles

Syntax に関して厳密に定義できるようになったので、今度は式がどのように評価されるかすなわち言語の意味の正確な定義が必要になる。意味の定式化には 3 種類の基本的な手法が存在する。

操作的意味 仮想的に term を解釈するマシンの動作を定義するもの。各 term とその term を解釈実行したときの次のマシンの状態 (または停止) を定義する。各 term の意味はマシンの状態の遷移と考えることができる。

表示的意味 単にマシンの状態ではなく、意味をもっと数学的・抽象的にとらえる。言語に表示的意味を与えるというのは、その言語における意味領域とそこでの解釈関数を定義することである。意味領域の探索は *domain theory* と呼ばれる、それはそれは深い研究分野として知られている。

公理的意味 プログラムの動作そのものを直接対象とするのではなく、動作の前後において成り立つ法則を扱う。この意味論はプログラムを推論する過程に焦点を当てており、不変量などという発想もこのあたりに起源をもつ。

60, 70 年代においては操作的意味論は他の二つに比べて劣るものと思われていた。それは、汚い言語仕様に適応しやすい分、洗練されておらず数学的にも弱かったからだ。しかし 80 年代に入り、より抽象的な方法は技術的困難に多くぶち当たるようになり、操作的意味論の単純さや柔軟さが比較的魅力的に見えるようになってきた。同時に、多くの研究者により表示的意味論で使用していた強力な数学的技法の多くが操作的意味論に転用できることとなり、脚光を浴びるようになってきた。