プログラム検証論と LISP プログラムミング

京都産業大学　理学部　計算機科学科
布施田　敏樹 (Yoshiki FUSEDA)

はじめに

このレポートは八杉 満利子教授指導の下で、特別研究としておこなった、プログラム検証論及び Lisp によるプログラミングについて書きました。
前半は、プログラム検証論について後半は、Lisp による和積標準形のプログラムとその解説について書きました。私の作ったプログラムは大変要領の悪い所、改良の余地がたくさんまあるかも知れませんが、時間が足りなかったためご了承ください。

目 次

1　形式的理論

この章では、ホーア論理の基礎をなす形式的理論について述べる。

1.1　形式的理論の言語

言語を定義するとは、変数、定数、関数記号、述語記号の４つの語彙を定義することである。そして、それらの四重対 ⟨ 変数, 定数, 関数記号, 述語記号 ⟩ をこの言語の語彙という。
ここでは、次の章で用いる自然数論の語彙 V^N を定義する。
変数 :$a, a0, a1, \cdots, b, b0, \cdots$
定数 :$0, 1$
関数記号 :$+, *$
述語記号 :$=, \leq$

次に語彙 V^N を使って自然数論の項と論理式を定義する。
1.変数と定数は項である。
2.t_1, t_2 が項ならば $+(t_1, t_2)$,$*(t_1, t_2)$ は項である。
これを BNF 記法で表せば次のようになる。

$$\text{項} :== \text{変数} \ \ |0 \ \ |1 \ \ |+(\text{項}, \text{項}) \ \ |*(\text{項}, \text{項}).$$

同様に論理式を BNF 記法で定義すれば

$$\text{論理式} ::= \leq (\text{項}, \text{項}) \ \ |= (\text{項}, \text{項}) \ \ |\neg(\text{論理式}) \ \ |(\text{論理式}) \wedge (\text{論理式}) \ \ |(\text{論理式}) \vee (\text{論理式}) \ \ |$$

となる。
語彙 V^N から定義された項の集合を $Term(V)$ と書き、語彙 V から定義された論理式の集合を $Fml(V)$ と書く。そして、この二つを合わせた対 $\langle Term(V), Fml(V) \rangle$ を語彙 V^N から定義される言語と呼ぶ。

1.2　形式的理論の意味論

前小節で V^N の言語を与えた。しかし、形式言語はあくまで記号列に過ぎず、それに意味を与えるには解釈 (interpretation) というものを考える。解釈の役割は、項と論理式に意味を与えることである。
語彙 V^N の (標準的解釈)I^N を定義する。
１.D=$\mathbf{N}$　$\mathbf{N}$ は自然数の集合を表す。(この D を解釈の領域と呼ぶ。)
２.$I^N(0) = 0, I^N(1) =$ １.
３.$I^N(+) = “+”, I^N(*) = “*”$.
４.$I^N(=) = “=”, I^N(\leq) = “\leq”$.

ただし“ $+$ ”,“ $*$ ”は自然数の和積を二変数関数と考えたもの、“ $=$ ” は自然数の等価関数と考えたものである。

語彙 V^Zの (標準的解釈)V^Nの定義。[1] １.D=$\mathbf{Z}$　$\mathbf{Z}$ は整数の集合を表す。
２.$I^N(Z) =$ “ $-$ ”.
３. 他は I^Nの場合と同じ。

次に項の解釈を定義する。それにはまず変数に与える環境というものを考える。環境ρというのは、変数の集合から領域 Dとする。そして項 t の I及びρのもとでの解釈 $I_\rho[t]$ を次のように定義する。

$$I_\rho[t] = \begin{cases} I[t] & t\text{ が定数のとき} \\ \rho[t] & x\text{ が変数のとき} \\ I[f]I_\rho[t_1], \ldots, I_\rho[t_n]) & t\text{ が } f(t_1, \cdots, t_n)\text{ のとき} \end{cases}$$

また、環境ρ、変数 x、Dの要素 a に対して、新しい環境を$\rho[a/x]$ を次のように定義する。

$$I_\rho[t] = \begin{cases} a & y\text{が変数 } x\text{ であるとき} \\ \rho(y) & \text{そのほかのとき} \end{cases}$$

この時次の項の代入性 (substitution　property) が成り立つ。

$$I_\rho[t_1[t_2/x]] = I_{\rho[I_\rho[t_2]/x]}[t_1]$$

項の解釈 (意味) が決まったので、今度は論理式の解釈を定義する。論理式の解釈とは、論理式の意味とは真理値のことである。限量子を除く論理記号 C に対するその解釈Ċを真理値から真理値を与える. . . $\forall$ と. . . $\exists$ を対応させ、関数として次のように定義する。

v_1	v_2	$\dot{\wedge}$	$\dot{\vee}$	$\dot{\Rightarrow}$	$\dot{\neg}$
True	**True**	**True**	**True**	**True**	**False**
True	**False**	**False**	**True**	**False**	**False**
False	**True**	**False**	**True**	**True**	**True**
False	**False**	**False**	**False**	**True**	**True**

限量子に対しては空でない真理値の集合 Sから真理値への関数として定義する。

S	$\dot{\forall}(S)$	$\dot{\exists}(S)$
$\{$**True**$\}$	**True**	**True**
$\{$**True, False**$\}$	**False**	**True**
$\{$**False**$\}$	**False**	**False**

---

[1] これは後の章でホーア論理の部分証明に使う。

語彙 I, 解釈 I及び環境ρのもとでの論理式 Fの意味 $I_\rho[F]$ は次のように定義する。

$$I_\rho[F] = \begin{cases} I[p](I_\rho[t_1], \cdots, I_\rho[t_n]) & F\text{が } P(t_1, \cdots, t_n) \text{ であるとき} \\ \dot{\wedge}(I_\rho[F_1], I_\rho[F_1]) & F\text{が } [F_1] \wedge [F_2] \text{ であるとき} \\ \dot{\vee}(I_\rho[F_1], I_\rho[F_1]) & F\text{が } [F_1] \vee [F_2] \text{ であるとき} \\ \dot{\Rightarrow}(I_\rho[F_1], I_\rho[F_1]) & F\text{が } [F_1] \Rightarrow [F_2] \text{ であるとき} \\ \dot{\neg}(I_\rho[F_1]) & F\text{が } [F_1] \text{ であるとき} \\ \dot{\forall}(\{I_{\rho[a/x]}[F_1] | a \in D\}) & \forall x.F\text{が } [F_1] \text{ であるとき} \\ \dot{\exists}(\{I_{\rho[a/x]}[F_1] | a \in D\}) & \exists x.F\text{が } [F_1] \text{ であるとき} \end{cases}$$

これらの定義のもとに論理式の正しさを定義する。
$\rho[A] = \mathbf{True}$ ならば、論理式 A は解釈 Iと環境ρに対して正しい (真である) といい、$I, \rho \models A$ と書く。任意の環境ρに対して $I, \rho \models A$ となる時、$I \models A$ と書き、論理式 A は解釈 Iに対して正しいという。さらに、任意の解釈 Iに対して、$I \models A$ が成り立っている時には、論理式 A は恒真論理式であるという。

1.3　形式的理論の証明

演繹に基づく数学的理論が存在する時、この少数の原理を公理系 (axiom system) という。この公理系を指定することが形式的理論を定義するということである。また形式的理論の公理系は、公理 (axiom) と呼ばれる論理式と、推論規則 (inference rule) と呼ばれるものを指定することによって定義される。

自然数の形式的理論　(形式的理論の語彙)

公理　(I 群)

I.1 $A \Rightarrow (B \Rightarrow A)$ ，

I.2 $(A \Rightarrow B) \Rightarrow (A \Rightarrow B \Rightarrow C) \Rightarrow A \Rightarrow C$ ，

I.3 $A \Rightarrow (B \Rightarrow (A \wedge B))$ ，

I.4a $(A \wedge B) \Rightarrow A$ ，

I.4b $(A \wedge B) \Rightarrow B$ ，

I.5a $A \Rightarrow (A \vee B)$ ，

I.5b $B \Rightarrow (A \vee B)$ ，

I.6 $(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow (A \vee B) \Rightarrow C$ ，

I.7 $(A \Rightarrow B) \Rightarrow (A \Rightarrow \neg B) \Rightarrow \neg A$ ， I.8 $\neg\neg A \Rightarrow A$，

I.9 $\forall x.A \Rightarrow A[t/x]$，

I.10 $A[t/x] \Rightarrow \exists x.A$，

I.11 $a = a$，

I.12 $a = b \Rightarrow A[a/x] \Rightarrow A[b/x]$．

公理（II 群）

II.N1 $a + 0 = a$，

II.N2 $a + (b + 1) = (a + b) + 1$，

II.N3 $a * 0 = 0$，

II.N4 $a * (b + 1) = a * b + a$，

II.N5 $a + 1 = b + 1 \Rightarrow a = b,$

II.N6 $\neg a + 1 = 0$，

II.N7 $A[0/x] \Rightarrow \forall x.(A \Rightarrow A[x+1/x]) \Rightarrow \forall x.A$，

II.N8 $a \leq b \Leftrightarrow \exists x.x + a = b$．

推論規則

推論規則は、すでに証明された論理式から新たな論理式を証明するための規則である。

$R1$

$$\frac{A \quad A \Rightarrow B}{B}$$ 三段論法

$R2$

$$\frac{C \Rightarrow A}{C \Rightarrow \forall\ x.A}$$ ただし変数 X は C の自由変数ではない

$R3$

$$\frac{A \Rightarrow C}{\exists x.A \Rightarrow C}$$ ただし変数 Xは Cの自由変数ではない

ただし、以上の公理、推論規則のうち A, B, Cは論理式を a, b, x は変数を、t は項を表す。

以上のように形式的理論の公理 I群と II群に分類される。公理 I群は、論理的公理と呼ばれるもので、すべての形式的理論に共通の公理である。

推論規則 $R1 - R3$ もすべて形式的理論に共通である。この共通の公理と推論

規則だけを持つ形式的理論を一階述語論理という。また、II群の公理は非論理的公理とよばれる。

次に形式的証明について定義する。
形式的証明とは、行番号、論理式、コメントの組みが並んだ図で次の二つの条件を満たす。
ただしコメントは、Thy^Nの公理の番号、あるいは推論規則の番号と行番号の有限列のことである。
１．i行目のコメントC_iが公理の番号である時、その行の論理式F_iは、その公理である。
２．i行目のコメントC_iが推論規則の番号と数の列R、$j_1, \cdots, j_m$であるときは、$j_1, \cdots, j_m < i$であり、

$$\frac{F_{j1}, \ldots, F_{jm}}{F_i}$$

が推論規則Rとなる。ただし、F_jはj行目の論理式のことである。
形式的証明によって論理式が証明できた時、論理式は形式的に証明可能であるという。

2 プログラム検証論 (Theory of Program Verification)

プログラム検証論とは、プログラム理論の主題の一つで、プログラムの正当性など、プログラムのもつ種々の性質の証明方法に関する理論をいう。実際の検証は、証明したい性質を形式的に記述して、形式的意味論に沿ったプログラム検証法を使うことにより行う。
目標 :
プログラム検証論は、数学レベルでのプログラムの検証を行うことにありプログラム検証論の数学的理論を作ることを目標にしている。

2.1 ホーア論理

プログラム検証論はフロイドによって始まりました。フロイドの方法は、流れ図の正しさを検証するもので直接プログラムを扱うものではなかった。フロイドの方法はホーアによってプログラムの正しさを直接検証する方法に改良される。

ホーア論理では

「プログラム Pは、仕様 Sを満たす」

という文章を表明付きプログラムというものを使って書きます。
表明付きプログラムというのは、プログラムの最初と最後に表明と呼ばれる条件を付加した形式のプログラムで

$$\{\mathrm{A}\}\ \mathrm{P}\ \{\mathrm{B}\},\langle\mathrm{A}\rangle\ \mathrm{P}\ \langle\mathrm{B}\rangle$$

と書き、意味は以下の通りである。A、B は表明と呼ばれる文章 (ドキュメント) であり、P はプログラムをしめす。

部分正当性 $\{\mathbf{A}\}\ \mathbf{P}\ \{\mathbf{B}\}$:プログラム P を条件 A が成立している状態で実行した時、停止するならば、その時条件 B が必ず成立する。

完全正当性 $\langle\mathbf{A}\rangle\ \mathbf{P}\ \langle\mathbf{B}\rangle$:プログラム P を条件 A が成立している場合で実行すると、実行は必ず成立し、成立後に条件 B が成立する。

プログラムが常に正しさを保つためには完全正当でなくてはならない。しかし、ホーア論理では部分正当性を考えることが多い。これは部分正当性が完全正当性よりも扱いやすいためである。完全正当性は部分正当性と停止性に分割することができる。ゆえに完全正当性は、部分正当性を検討してから停止性の条件を付加すればよい。

2.2 部分正当性のホーア論理

ホーア論理では、部分正当性と完全正当性の2種類の理論があるので、その形式的理論も部分正当性の形式的理論と完全正当性の形式的理論の2種類がある。ここでは、[1] にならって部分性正当性を PHL、完全正当性を THL と書くことにする。

PHL は一つだけ定義するのではなく、二つの語彙 PV,AV と語彙 AV 上の論理式 S を決めるごとに、一つの形式的理論 $PHL(PV, AV, S)$ を定義する。PV はプログラムの語彙であり、AV は表明、すなわち論理式の語彙である。また論理式の集合 S は帰結規則の前提条件にある二つの「正しい表明」を指定するために使う。プログラムの語彙、表明の語彙は一階の形式的理論である。

ホーア論理の形式的理論を定義するためには、プログラムの語彙、表明の語彙 AV を語彙とする論理式の集合 $Fml(AV)$ の部分集合 S を選ぶ。そして、この S を非論理的公理の集合とみなす。ホーア論理では。表明付きプログラムを証明する体系であるが、非論理的公理は表明である。

ホーア論理の形式的理論 $PHL(PV, AV, S)$ を語彙 PV,AV と集合 S から定義する。この理論の形式言語は、表明、プログラム、表明付きプログラムからなり、プログラムの定義はターゲットにするプログラミング言語によって異なるが、ここでは [1] にならって **Min** という while プログラムの言語だけ考える。

Min は、プログラムの語彙 PV にだけ依存し、表明の語彙 AV や非論理的公理の集合 S には依存しない。そこで、語彙 PV から定義された **Min** を **Min**(V) と書くことにする。**Min**(V) の文法は以下のように定義する。

プログラミング言語 **Min**(V)

```
    変数 ::=  語彙 PV の変数 .       PV は形式的理論

      式 ::=  Term(PV) の項 .

  ブール式 ::=  Fml(PV) の原子論理式  |ブール式 ∧ ブール式  |
              ブール式 ∨ ブール式 |ブール式 ⇒ ブール式  |
              ¬ ブール式 .

      文 ::=  skip |変数 ::= 式  | begin 文 {" ;"} end  |
              if ブール式 then 文 else 文 fi  |
              while ブール式 do 文 od .
```

プログラム::= 文 .

プログラムが定義できたの、次は表明と表明付きプログラムを定義する。

部分正当性の表明と表明付きプログラム

表明::= $Fml(AV)$ の論理式．

表明付きプログラム::= “ $\{$ ” 表明 “$\}$” プログラム “ $\{$ ” 表明 “$\}$” ．

以上でホーア論理の形式言語が定義できたので今度は形式的証明を定義する。
部分正当性の形式的理論の公理と推論規則

公理

A_{as} （代入文の公理）

$$\{\mathrm{A[t/x]}\}\ \mathrm{x := t}\ \{\mathrm{A}\}$$

A_{sk} （スキップ文の公理）

$$\{\mathrm{A}\}\ \mathbf{Skip}\ \{\mathrm{A}\}$$

推論規則

R_{if} （条件文の規則）

$$\frac{\{C \wedge A\}\ P\ \{B\} \quad \{\neg C \wedge A\}\ Q\ \{B\}}{\{A\}\ \mathbf{if\ C}\ \{\mathbf{then}\}\ \mathbf{P\ else\ Q\ fi}\ \{\mathbf{B}\}}$$

R_{wh} （while 文の規則）

$$\frac{\{C \wedge A\}\ P\ \{A\}}{\{A\}\ \mathbf{while}\ C\ \mathbf{do}\ P\ \mathbf{od}\ \{\neg C \wedge A\}}$$

R_{cp} （複合文の規則）

$$\frac{\{A\}P_1\{S_1\}...\{S_{i-1}\}P_i\{S_i\}...\{S_{n-1}\}P_n\{B\}}{\{A\}\ \mathbf{begin}\ P_1\ ;...\ \ ;P_n\ \mathbf{end}\ \{B\}}$$

R_{cs} （帰結規則）

$$\frac{S \vdash A \Rightarrow B\ \{B\}P\{C\}\ S \vdash C \Rightarrow D}{\{A\}P\{D\}}$$

ホーア論理の形式的証明
$PHL(PV, AV, S)$ の証明可能性は、表明の形式的理論における証明可能性と同じように定義する。$PHL(PV, AV, S)$ の形式的証明とは、行番号、表明付きプログラムコメントからなる行を並べたもので、次の条件を満たすものをいう。

1 i 行目のコメント C_iが公理の番号である時、その行の表明付きプログラム AP_iはその公理である。

2 i 行目のコメント C_iが推論規則の番号と数の列 R $J_i, \cdots, J_m$である時は $J_1, \cdots, J_m < i$ であり

$$\frac{AP_{j1}, \cdots, AP_{jm}}{AP_i}$$

が、その推論規則 Rとなる。

2.3　ホーア論理によるプログラムの証明

以上で定義したホーア論理の公理及び推論規則を使って、以下のユークリッド互除法による GCD のプログラムの部分証明を行う。

GCD を求めるプログラム
＜整数 x、y の少なくとも一方がゼロではない＞

```
begin
a:=x;  b:=y;
while b≠ 0 do
  begin
    a:= a mod b;
    c:= a;a:= b;b:= c
  end
if a < 0  then a:= -a fi
end
```

＜ a は x と y の最大公約数である。＞

while b≠ 0 do ... end の... を 4 つの部分的プログラムに分けて証明を行なってみる。

1、 a:= a mod b
2、 c:= a
3、 a:= b
4、 b:= c

While 文の部分プログラムの証明 ：

A1 ≡ ¬ x = 0 ∨ ¬ y = 0
A2 ≡ A1 ∧ ¬ a ∨ ¬ b = 0

1 {¬ b = 0 ∧ A2 ∧ gcd(a mod b,b) = gcd(x,y)} a:= a mod b
{¬ b = 0 ∧ A2 ∧ gcd(a mod b,b) = gcd(x,y)} (Aas)

2 {¬ b = 0 ∧ A2 ∧ gcd(a,b) = gcd(x,y)} a:= a mod b
{A2 ∧ gcd(a,b) = gcd(x,y)} (Rcs,1)

3 {A2[c/b] [b/a] [a/c] ∧ gcd(a,b) = gcd(x,y)} c:=a
{A2[c/b] [b/a] ∧ gcd(c,b) = gcd(x,y)} (Aas)

```
4 {A2               ∧  gcd(a,b) = gcd(x,y) } c:=a
  {A2[c/b] [b/a]  ∧  gcd(c,b) = gcd(x,y) }                         (
Rcs,3 )

5 {A2[c/b]  [b/a]  ∧  gcd(c,b ) = gcd(x,y) } a:=b
  {A2[c/b]  ∧  gcd(c,a) = gcd(x,y) }                               (
Aas )

6 {A2[c/b]      ∧  gcd(c,a)  = gcd(x,y) } b:=c
  {A2  ∧  gcd(b,a) = gcd(x,y) }                                    (
Rcs,5 )

7 {¬ b  =  0  ∧  A2  ∧  gcd(a,b)  =  gcd(x,y) }
      begin
         a: =  a mod b  ;
         c:=a  ;a:=b  ;b:=c  ;
      end
  {A2  ∧  gcd(b,a) = gcd(x,y) }                                    (
Rcp 2,4,5,6 )

8 {¬ b  =  0  ∧ A2  ∧  gcd(a,b) = gcd(x,y) }
     begin
         a: =  a mod b  ;
         c:=a  ;a:=b  ;b:=c  ;
      end
   {A2  ∧  gcd(a,b) = gcd(x,y) }                                  (Rcs,7)

9 {A2  ∧  gcd(a,b) = gcd(x,y) }
    while  ¬ b=0  do
      begin
       a:= a mod b;
       c:= a;  a:= b;  b:= c
      end
    od
   { ¬ ¬ b=0  ∧  A2  ∧  gcd(a,b) = gcd(x,y) }                     (Rwh,8)
```

プログラムの部分証明の概略図

$$\cfrac{\cfrac{1(A_{as})}{2}\;(R_{cs},1) \qquad \cfrac{3(A_{as})}{4}\;(R_{cs},3) \qquad 5(A_{as}) \quad 6(A_{as})}{\cfrac{\cfrac{7}{8}\;(R_{cs},7)}{9}\;(R_{wh},8)}\;(R_{cp}2,4,5,6)$$

形式的証明の説明 ：

この証明の語彙は V^Z に関数 mod を追加したものである。この語彙 PV_1 と書くことにする。ただし、$mod(a,b)$ は $a \ \mathbf{mod} \ b$ のことである。

表明の語彙は、PV_1 に、関数記号 gcd, abs、そして述語記号“ $<''$を追加したものである。この語彙を AV_1 と書くことにする。

非論理的公理は、公理となる論理式を並べるのではなく、解釈から定義する。表明の解釈は V^Z の標準的解釈 I^Z を AV_1 に拡張し、解釈 I_1 を定義する。I_1 では mod は割算の余りを、gcd は最大公約数、abs は絶対値を計算する関数、そして“ $<''$は大小関係で解釈する。

解釈 I_1 の定義

$$I_1(\mathbf{mod}(a,b)) = \begin{cases} 0 & b = 0 \text{ であるとき} \\ a \text{ を } b \text{ で割った余り} & \text{それ以外のとき} \end{cases}$$

$$I_1(gcd(a,b)) = \begin{cases} 0 & a = b = 0 \text{ であるとき} \\ a \text{ と } b \text{ の最大公約数} & \text{それ以外のとき} \end{cases}$$

$$I_1(abs)(a) = a \text{ の最大公約数}$$

$$I_1((<)(a,b)) = \begin{cases} \mathbf{True} & a < b \text{ であるとき} \\ \mathbf{False} & \text{それ以外のとき} \end{cases}$$

この解釈 I_1 がで正しい論理式の集合を S_1 と

$$S_1 \equiv \{F | I_1 \models F\}$$

になる。

GCD プログラムの形式的証明は、ホーア論理 $PHL(PV, AV, S)$ 拡張 $PHL(PV_1, AV_1, S_1)$ の形式的証明になっている。

2.4 完全正当性のホーア論理

これまでは部分正当性について考えてきたが、今度は、完全正当性の場合について考えてみる。完全正当性というのは部分正当性＋停止性であったから、部分正当性のホーア論理を完全正当の部分に拡張するには、停止性が問題となる。$while$ プログラムの言語である **Min** に限っていえばプログラムが停止しない可能性があるのは、$while$ 文の規則が問題となる。

部分正当性の場合は、非論理的公理 Sを指定して $PHL(PV, AV, S)$ を定義したのであった。しかし、完全正当性の場合には、非論理的公理 Sを指定する $PHL(PV, AV, S)$ という形では、正しい形式的理論を定義できない。

そのためまず解釈 Iを指定して $PHL(PV, AV, I)$ を定義する。次に完全正当性の形式的理論 $THL(PV, AV, I)$ を定義する。THL を定義するには、PHL の $while$ 文を変更するだけでよい。そのためにはまず整礎表明というものを定義する。

任意の集合 Bの要素を引数とする 2 項関係 $W(x, y)$ が整礎であるとは、x_0, ... ,x_i,... という Bの要素の無限列で $W(x_{i+1}, x_i)$ となるものが無いことをいう。

例えば、$\cdots x_i < x_{i+1} < \cdots < x_1 < x_0$となる自然数の無限列 $x_0, \cdots, x_i, \cdots$ を作ることはできないから、自然数を引数とする。 2 項関係 $x < y$は整礎である。しかし、$x < y$を整数を引数とする 2 項関係とみなせば $\cdots < -2 < -1 < 0$ という数列は作れるので整礎でなくなる。関係 $W(x, y)$ を 「x が y より小さい」読めば、Wが整礎であるとは、いくらでも小さくなるような要素の無限列は存在しないということである。

整礎表明とは、この整礎関係を表現するような表明をいう。

表明 Rの自由変数がちょうど $p_1, \cdots, p_n, p'_1, \cdots, p'_n$である。このとき、表明 R が (この変数列に対して) 整礎表明であるとは、

$$W(\langle v'_1, \ldots, v'_n \rangle, \langle v_1, \ldots, v_n \rangle) \iff (I, \rho_0 \models R)$$

と定義した 2 項関係 Wが整礎となることでいう。ただし、$v'_1, \cdots, v'_n, v_1, \cdots, v_n$ は解釈 Iの領域の要素であり、$\langle v'_1, \cdots, v'_n \rangle$ と $\langle v_1, \cdots, v_n \rangle$ はその n 重対である。またρ_0は

$$\rho_0 = v_i \ , \ \rho_0(p'_i) = v_i \quad (i = 1, \cdots, n)$$

となる環境である。この条件を満たすρ_0は一つではないが、Rの自由変数がちょうど $p_1, \cdots, p_n, p'_1, \cdots, p'_n$なので、どの$\rho_0$を選んでも良い。たとえば、解釈 Iが大小関係 を含む自然数の語彙の標準的解釈ならば、表明 $p'_i \le p_1$は整礎表明である。同じことであるが、$p'_i + 1 \le p_1$も整礎表明になる。しかし $p_1 \le p'_1$は整礎表明ではない。また解釈 Iが整数の語彙の標準的解釈である場合には、表明 $p'_1 \le p_1$は整礎でなくなる。しかし $p'_1 \le_N p_1 \equiv p'_1 \le p_1 \wedge \le p'_1$ と定義すれば表明 $p'_1 \le_N p_1$は整礎となる。以上の例では、変数 p_iと p'_iが

一つずつであったが、これが複数となる場合としては $p'_1 + p'_2 \le p_1 + p_2$ や $p'_1 \le p_1 \vee p'_1 = p_1 \wedge 0p'_2 \le p_2$ などがある。ただし、いずれも自然数の表明である。

整礎関係を使って完全正当性のホーア論理 $THL(PV, AV, S)$ を定義する。$THL(PV, AV, S)$ は $while$ 文以外の推論規則は $PHL(PV, AV, I)$ と同じであり、$while$ 文の規則のみを $\mathtt{R}_{\mathtt{twh}}$ という規則で置き換える。その規則は整礎関係を必要とするのが、整礎関係は解釈 I を指定して始めて意味を持つ概念なので、この推論規則の定義には解釈が欠かせない。そのために $THL(PV, AV, S)$ が定義できない。そこで $THL(PV, AV, I)$ を直接定義することになる。

$THL(PV, AV, I)$ の公理と推論規則　$THL(PV, AV, I)$ の公理と $while$ 文以外の推論規則は $THL(PV, AV, I)$ と同じである。そして、$while$ 文の規則 $\mathtt{R}_{\mathtt{wh}}$は、次の規則 $\mathtt{R}_{\mathtt{twh}}$に変更する。

$\mathtt{R}_{\mathtt{twh}}$ ：

$$\frac{\langle (t_1 = p_1) \wedge ... \wedge (t_n = p_n) \wedge C \wedge A\rangle\ P\ \langle R[t_1/p'_1, ..., t_n/p'_n] \wedge A\rangle}{\langle A\rangle \mathbf{while}\ C\ \mathbf{do}\ P\ \mathbf{od}\ \langle \neg C \wedge A\rangle}$$

ただし、$t_1, \cdots, t_n$は項である。この項を $while$ 文の推論規則の変量（**variant**）と呼ぶことにする。また、次の二つの付帯条件が成り立つとする。

１.$p_1, \cdots, p_n, p'_1, \cdots, p'_n$は、表明 A、Cにも、項 $t_1, \cdots, t_n$にも、また、プログラム Pにも現れない変数である（つまり、R以外にも現れない）。

２.Rは変数列 $p_1, \cdots, p_n, p'_1, \cdots, p'_n$ に対して整礎表明である。

この規則が完全正当性を保証する説明 ：
簡単のために $n = 1$ として説明する。まず、$while$ 文の規則の前提が成り立つと仮定する。状態 p が $t_1 = p_1 \wedge C \wedge A$ という前条件を満たすときに Pを実行すると $R[t_1/p'_1]$ という後条件が成り立っている。このとき A が成り立つので、これがループ不変表明である。したがって、$while$ ループの停止性さえ証明できればよい。停止性を証明するには、実行後に $R[t_1/p'_1]$ が成り立っていることを利用する。まず、実行前に変量 t_1 の値が v_1であったとする。そして、Pを実行したとする。第二の付帯条件から p_1の値は Pの実行前後で不変なので、実行後の t_1の値を v'_1とすると、$I, \rho_0 \models R$となる。ただし、ρ_0は $\rho_0(p_1) = v_1, \rho_0(p'_1) = v'_1$となる環境である。よって、$W$の定義から $W(v'_1, v_1)$ が満たされる。$while$ ループ無限に実行されると、同様にどんどん「小さく」なる無限列 v_1, v'_1, v''_1 ができる。ところが、Wは整礎関係だから、これらは矛盾である。よって、$while$ ループ文は終了する。
以上で完全正当性の定義ができた。

3　ホーア論理 (プログラム検証論) の現実的限界

先ほどのユークリッド互除法によるプログラムの部分証明からもわかるように小さなプログラムを証明するのも大変である。
ホーア論理は

1 仕様記述と証明の仕様コストが膨大である。

2 証明にバグがないと保証はない。

3 仕様にバグばないという保証はない。

という特徴をもつ。これらの問題は、ホーア論理における特殊な問題ではなく、プログラム検証論全体の問題である。このような限界のため、1980年代には、完全なプログラム検証論は難しいという意見が支配的でした。しかし現在では、形式的技法と総称されるより現実的な方法が開発され、徐々に実用が始まりつつあります。

4　形式的技法へ

形式的技法とはソフトウェアの開発において、記号論理学 (形式的論理学) から生まれた技法を利用することです。

形式的技法の最低条件は、形式的仕様記述です。しかし、これは最低条件であり証明をないがしろにするわけではありません。形式的技法では、必要なだけ、可能な範囲で正しさの証明を行います。つまり、完全な正さの証明ができない場合には、信頼性の向上を目的として、部分的な正さの証明を行うのです。形式的技法は、バグを根絶する技術ではなくバグの危険性を低減する技術なのです。
この定義からすると、ホーア論理はまさに形式的技法の一つといえ、「ホーア論理のようなプログラム検証論」が形式的技法の源流であり、現在もその基礎であることはかわりがありません。

5 LISP (LISt Processor)

リスト処理、記号処理のために設計されたプログラミング言語およびそれらの言語処理系の総称。Lispの名はLISt ProcessorあるいはLISt Processingの略である。人工知能用プログラム言語として1950年代末から1960年代初めにかけて米国マサチューセッツ工科大学(MIT)のJ.Mc.Carthy教授が考案したプログラム言語。この最初のLispは、今日純Lispとよばれている。その後、応用分野の拡大と計算機技術の向上に伴い、数多くのLispの方言とそれらの処理系が開発されてきた。アメリカで開発されたLisp1.5,Interlisp,MacLisp,わが国で開発されたUTILisp Lisp1.9などが有名である。[2]

LISPの特徴 ：
リストオブジェクトはコンスかアトムである。アトムはもっとも基本的なオブジェクトです。またLispで扱うものはすべてリストの形をしている。Lispプログラムはいくつかの関数を構成単位とし、再帰呼び出しを含む関数呼び出しを制御の基本とします。変数や関数の型宣言をする必要がありません。プログラムは (関数 引数) の形で、関数を引数に適用する形になっている。ほかの言語でも再帰はありますが、ループで書く場合には、再帰ということは表にでません。あとラムダ(λ)という都合のよい表現法がある。これにより、関数にいちいち名前をつけなくても、ひとつのプログラムの中では関数として使える。

例

```
(defun my-function (引数)
       .
       .
       .
   (my-function ...
       .
       .
       .
         ))
```

6　和積標準形　(Product of Sum)

和積標準形とは　:

命題変数 (x) またはその否定　(¬x) の形の式をリテラル、リテラルから∨(OR) のみで作られる式を基本和、基本和から∧(and) のみで作られる式を和積標準形という。

例 $(\neg X \vee Y \vee Z) \wedge (X \vee \neg Z) \wedge (\neg Y \vee Z)$
(ただし　:¬を not、∧を and、∨を or とします。)
リテラル X、$\neg X$、Y、$\neg Y$、Z、$\neg Z$
基本和 $(\neg X \vee Y \vee Z)$、$(X \vee \neg Z)$、$(\neg Y \vee Z)$
和積標準形 $(\neg X \vee Y \vee Z) \wedge (X \vee \neg Z) \wedge (\neg Y \vee Z)$

[命題]
任意の式を次の手続きで、同値な和積標準形に変形する。

和積標準形への手続き [3]
(一理論であり他にも多数変形の手続き手順あり)

(1)$A \Rightarrow B$を$\neg A \vee B$で置き換える。
(2)$\neg\neg A$ を A で置き換える。
(3)$\neg(A \vee B)$ を$\neg A \wedge \neg B$で置き換える。
(4)$\neg(A \wedge B)$ を$\neg A \vee \neg B$で置き換える。
(5)$A \vee (B \wedge C)$ を $(A \vee B) \wedge (A \vee C)$ で置き換える。
(6)$(A \wedge B) \vee C$を $(A \vee C) \wedge (A \vee B)$ で置き換える。

例　:$(X \Rightarrow (Y \wedge Z)) \wedge ((\neg Y) \Rightarrow (X \wedge Y))$ の和積標準形を求める。
　$= (\neg X \vee (Y \wedge Z)) \wedge (\neg\neg Y \vee (X \wedge Y))$　(1) を二回適用
　$= (\neg X \vee (Y \wedge Z)) \wedge (Y \vee (X \wedge Y))$　(2) 適用
　$= (\neg X \vee Y) \wedge (\neg X \vee Z) \wedge (Y \vee X) \wedge (Y \vee Y)$　(5) 適用 ... Answer
Not Tautology　(恒真な式かどうか判定する。)

使用例
○　任意の式を和積標準形に直すことができるので恒真性の判定に使えます。
○　和積標準形は、論理回路の効率的設計に応用できます。 (命題論理と論理回路は対応します。)

このプログラムは中置記法を前置記法に直すプログラムです。ただし (A and B or C) では正しい答えは返ってきません。((A and B) or C) または (A and (B or C)) のように書き換えてください。

```
>(defun prefix (exp)
     (cond ((atom exp) exp)
           ((or (eq 'and (second exp))
                (eq 'or (second exp))
                (eq '=> (second exp)))
            (list (second exp) (prefix (first exp))
                               (prefix (third exp))))
           ((eq 'not (first exp))
            (list (first exp) (prefix (second exp))))
           (T 'nil)))
prefix
```

例

```
> (prefix '((a and b) or c))
(OR (AND A B) C)
> (prefix '((not x) => ((a and b) or c)))
(=> (NOT X) (OR (AND A B) C))
```

このプログラムは implication(⇒) を NOT OR の形に書き換えるプログラムです。

```
>(defun implication (exp)
        (cond ((atom exp) exp)
              ((eq '=> (car exp))
               (list 'or (list 'not (implication (second exp)))
                                    (implication (third exp))))
              ((eq 'or (car exp))
               (list 'or (implication (second exp))
                         (implication (third exp))))
              ((eq 'and (car exp))
               (list 'and (implication (second exp))
                          (implication (third exp))))
              ((eq 'not (car exp))
               (list 'not (implication (second exp))))
              (T 'nil)))
implication
```

例

```
> (implication '(=> x y))
(OR (NOT X) Y)
> (implication '(=> (=> a b) c))
(OR (NOT (OR (NOT A) B)) C)
```

このプログラムは double negation を処理するプログラムです。

```
>(defun mynot (exp)
      (cond ((atom exp) exp)
            ((and (eq 'not (car exp))
                  (atom (cadr exp)))
             exp)
            ((and (eq 'not (first exp))
                  (eq 'not (first (second exp))))
             (mynot (second (second exp))))
            ((and (eq 'not (first exp))
                  (eq 'or (first (second exp))))
             (list 'and (mynot (list 'not (second (second exp)))
                        (mynot (list 'not (third (second exp))))))
            ((and (eq 'not (first exp))
                  (eq 'and (first (second exp))))
             (list 'or (mynot (list 'not (second (second exp))))
                       (mynot (list 'not (third (second exp))))))
            ((eq 'or (car exp))
             (list 'or (mynot (second exp)) (mynot (third exp))))
            ((eq 'and (car exp))
             (list 'and (mynot (second exp)) (mynot (third exp))))
            (T 'nil)))
mynot
```

例

```
> (mynot '(not (not (not (not a)))))
A
> (mynot '(not (not (or a b))))
(OR A B)
```

このプログラムは分配法則に関するプログラムです。(A or (B and C)) を (A or B) and (A or C) に変換し ((A and B) or C)) を (A or C) and (B or C) に変換します。

```
(defun subpos (exp)
  (cond ((atom exp) exp)
        ((eq ’not (first exp))
         (list ’not (subpos (second exp))))
        ((eq ’and (first exp))
         (list ’and (subpos (second exp)) (subpos (third exp))))
        ((and (eq ’or (first exp))
              (atom (second exp)))
           (cond ((atom (third exp))
                (list ’or (subpos (third exp)) (subpos (second exp))))
               ((eq ’not (first (third exp)))
                (list ’or (subpos (second exp)) (subpos (third exp))))
               ((eq ’or (first (third exp)))
                (list ’or (subpos (second exp)) (subpos (third exp))))
               ((eq ’and (first (third exp)))
                (list ’and (subpos (list ’or (subpos (second exp))
                                             (subpos (second (third exp)))))
                           (subpos (list ’or (subpos (second exp))
                                             (subpos (third (third exp)))))))
                (T (subpos (list ’or (subpos (third exp))
                                     (subpos (second exp)))))))
        ((and (eq ’or (first exp))
              (eq ’not (first (second exp))))
           (cond ((atom (third exp))
                  (list ’or (subpos (third exp)) (subpos (second exp))))
                 ((eq ’not (first (third exp)))
                  (list ’or (subpos (second exp)) (subpos (third exp))))
                 ((eq ’or (first (third exp)))
                  (list ’or (subpos (second exp)) (subpos (third exp))))
                 ((eq ’and (first (third exp)))
                  (subpos (list ’or (subpos (third exp))
                                    (subpos (second exp)))))
                (T (subpos (list ’or (subpos (third exp))
                                     (subpos (second exp)))))))
        ((and (eq ’or (first exp))
```

```
              (eq 'or (first (second exp))))
              (cond ((atom (third exp))
                (list 'or (subpos (third exp)) (subpos (second exp))))
               ((eq 'not (first (third exp)))
                (list 'or (subpos (second exp)) (subpos (third exp))))
               ((eq 'or (first (third exp)))
                (list 'or (subpos (second exp)) (subpos (third exp))))
               ((eq 'and (first (third exp)))
                (list 'and (subpos (list 'or (subpos (second exp))
                                             (subpos (second (third exp)))))
                           (subpos (list 'or (subpos (second exp))
                                             (subpos (third (third exp)))))))
                (T (subpos (list 'or (subpos (third exp))
                                     (subpos (second exp)))))))
        ((and (eq 'or (first exp))
              (eq 'and (first (second exp))))
         (list 'and (subpos (list 'or (subpos (third exp))
                                      (subpos (second (second exp)))))
                    (subpos (list 'or (subpos (third exp))
                                      (subpos (third (second exp)))))))
       (T 'nil)))

SUBPOS
```

例

```
> (subpos '(or a (and b c)))
(AND (OR B A) (OR C A))
> (subpos '(or (and a b) c))
(AND (OR A C) (OR B C))
```

このプログラムは前置記法を中置記法に変換するプログラムです。ただし(and or A B C)では正しい答えは返ってきません。(or (and A B) C)または(and A (or B C))のように書き換えてください。

```
>(defun center (exp)
     (cond ((atom exp) exp)
           ((or (eq 'and (first exp))
                (eq 'or (first exp))
                (eq '=> (first exp)))
            (list (center (second exp)) (center (first exp))
                                        (center (third exp))))
```

```
           ((eq 'not (first exp))
            (list (first exp) (center (second exp))))
           (T 'nil)))
CENTER
```

例

```
> (center '(and a (or a b)))
(A AND (A OR B))
> (center '(or (and a b) (or a c)))
((A AND B) OR (A OR C))
```

このプログラムは任意の論理式の恒真性を判定するSUBプログラムです。

```
>(defun wang (l la r ra)
  (if (null l)
      (if (null r)
       (if (intersect la ra)
           (format T "~% Tautology~%")
           (format T "~% Not Tautology~%"))
        (let ((r0 (car r)) (r1 (cdr r)))
           (if (atom r0)
              (wang l la r1 (cons r0 ra))
              (case (car r0)
                (not (wang (cons (cadr r0) l) la r1 ra))
                (and (and (wang l la (cons (cadr r0) r1) ra)
                          (wang l la (cons (caddr r0) r1) ra)))
                (or (wang l la
                     (cons (cadr r0) (cons (caddr r0) r1)) ra))
                ))))
     (let ((l0 (car l)) (l1 (cdr l)))
        (if (atom l0)
             (wang l1(cons l0 la) r ra)
               (case (car l0)
                 (not (wang l1 la (cons (cadr l0) r) ra))
                 (and (wang (cons (cadr l0)(cons (caddr l0) l1)) la
                       r ra))
                 (or (and (wang (cons (cadr l0) l1) la r ra)
                          (wang (cons (caddr l0) l1) la r ra))))))))
WANG
```

例

```
> (wang nil nil '(and a (or a b)) nil)
 Not Tautology

NIL
> (wang nil nil '(or a (or (not a) (not b))) nil)
 Tautology

NIL
```

恒真性を判定するプログラムの補助関数です。

```
>(defun intersect (x y)
     (if (null x)
        nil
        (if (memq (car x) y)
           t
           (intersect (cdr x) y))))
INTERSECT
```

恒真性を判定するプログラムの補助関数です。

```
>(defun memq (el lst)
        (member el lst :test #'eq))
MEMQ
```

恒真性を判定するプログラムの Main です。
ただし (A and B or C) では正しい答えは返ってきません。
((A and B) or C) または (A and (B or C)) のように書き換えてください。

```
>(defun judge (exp)
       (setq formula (subpos (mynot
                      (implication (prefix exp)))))
       (wang nil nil formula nil))
JUDGE
```

例

```
> (judge '(((not a) => (a or b))))
 Not Tautology

NIL

> (judge '((a => b) => (a => (b => c)) => (a => c)))
 Tautology

NIL
```

このプログラムは、今までの各プログラムを組み合わせて論理式を代入すると、和積標準形と恒真性を判定するができます。

```
>(defun pos (exp)
       (setq answer (center (subpos
                    (mynot  (implication
                    (prefix exp))))))
        (format T "~% ~S ~%" answer)
       (judge exp))
POS
例
> (pos '((a => b) => (a => (b => c)) => (a => c)))
 ((A OR ((NOT A) OR (C OR (NOT B)))) AND
 (((NOT A) OR (C OR (NOT B))) OR (NOT B)))

 Tautology

NIL
> (pos '(((not a) => (a or b)) and (not a)))
 ((A OR (B OR A)) AND (NOT A))

 Tautology

NIL
```

このプログラムはユークリッド互除法 (第三章) による 2 変数整数の GCD を求めることができます。

```
>(defun mygcd (x y)
  (setq a x)(setq b y)
    (cond ((not(zerop b))
           (setq a (mod a b))
           (setq c a)(setq a b)(setq b c)
           (mygcd a b))
          (if (< a 0) (abs a) a )))

MYGCD
例
>(mygcd '12 '24)
2
```

```
>(mygcd '-12 '-48)
12
```

参考文献

[1] 林 晋 プログラム検証論 共立出版株式会社 情報数学講座 8 1995

[2] RODNEY A.BROOKS 井田 昌之訳 Common Lisp プログラミング 丸善株式会社 1986

[3] 八杉 満利子 林 晋 情報系の数学入門 オーム社 1993

[4] 萩谷 昌巳 ソフトウェア科学のための論理学 岩波講座ソフトウェア科学 11 岩波書店

[5] 岩波情報科学辞典 岩波書店 1990

7　最後に

本研究の全過程を通じて、御教授、御指導を頂きました八杉 満利子教授には心より感謝致します。また、本研究において色々と助言をしていただいた、八杉研究室の院生の中田昌宏さん、実藤法道さんにも感謝致します。