Logical Program Synthesis の Implementation (pilot1) について

琴野 実, 大村伸一, 宮沢君江 (京産大・理)
謝 章文 (京産大・計科研)

I. まえがき

LPS (Logical Program Synthesis) とは, formal specification から正当性と実行可能性の保証された program を構成する過程で, 演えき体系における導出可能性や定義可能性に関する機能を利用する program 合成の方法のことである。(参考文献 [1],[2],[3],[4])

謝[1]は, 1977年11月のソフトウエア工学研究会で, Resolution-Refutation 法に基づく logical calculus の拡張であるいくつかの LPS calculi を示し, それらの calculi を用いて, 関数と等号を含む一階述語論理の言語を記述言語とする formal specification から, recursive type program を合成する LPS procedure を与えた。さらに, LPS procedure の正当性を保証するために, Herbrand Theory と a theory of meaning and interpretation, 特に a theory of designation に基づく LPS の基礎理論をも示した。

ここでは, LPS の簡単な紹介と, その研究会での LPS procedure のデモンストレーション用に開発された pilot1 の機能, そのシステム構成, および pilot1 implementation 時の作業環境, 条件について述べる。

pilot1は, LPS 理論の例証を行う目的で, 上述の研究会におけるデモンストレーション用として作成され, Lisp の基本関数から, evalquote 程度の program の合成が行えることを目途とした。

また, これは, その名の示すように, 京産大計算機科学研究所の LPS project (上村, 謝) の一環として作成された, 実験的システムである。

II. LPS procedure.

pilot1で用いた, formula の合成機能を含む LPS calculus K_1 による LPS procedure を, Lisp function equal の合成を例に説明する。

calculus K_1 は, 謝[1]において提示された LPS calculus のうち最も基本的なものである。

Lisp function equal の formal specification (spec) を与える。

equal の spec

conjecture:
1. P(x,y,z)
入力変数: x,y, 出力変数: z.
axioms:
2. $\neg\Pi(\mathrm{atom}[u])\vee\neg\Pi(\mathrm{atom}[v])\vee P(u,v,\mathrm{eq}[u,v])$
3. $\neg\Pi(\mathrm{atom}[u])\vee\Pi(\mathrm{atom}[v])\vee P(u,v,\mathrm{false})$
4. $\Pi(\mathrm{atom}[u])\vee\neg\Pi(\mathrm{atom}[v])\vee P(u,v,\mathrm{false})$
5. $\Pi(\mathrm{atom}[u])\vee\Pi(\mathrm{atom}[v])\vee\neg P(\mathrm{car}[u],\mathrm{car}[v],w)\vee\neg P(\mathrm{cdr}[u],\mathrm{cdr}[v],k)\vee P(u,v,\mathrm{and}[w,k])$
IDaxiom (Implicite Definition axiom):
6. P(u,v,equal[u,v])
出力関数: {equal}
executable predicate: {atom, eq, and}

ここで

. conjecture は, equal の入出力関係を示し, equal[x,y]=z を表わしている。conjecture に含まれる変数は, 入力変数と出力変数とに区別される。それ以外の変数は, 中間変数と呼ばれる。

axioms は, equal に関する事実の記述である。各 clause は, つぎのように解釈される。

2: u,v が共に atom であれば, equal の値が求める値である。

3,4: u,v の一方が atom であり, 他方がそうでない場合, 値は false である。

5: u,v が共に atom でない場合には, car[u] と car[v] に対する equal の値と,

cdr[x] と cdr[y] に対する equal の値との論理積(and)である。

IDaxiom は，一般に conjecture の出力変数を，それに対応する出力関数と入力変数から作られる表現(equal[x,y]) の variant である。これは，recursive program の合成に必要とされるものである。

LPS では，(条件付きの)term の合成を行うので，この spec に見られるように，eq, and, equal などの Lisp predicate は，true or false の値をとる logical function とみなされる。logical function を predicate として用いるためには，つぎのような predicate Π を使う。

Π(f[x]): 真 ⟺ f[x]=true
Π(f[x]): 偽 ⟺ f[x]= false.

Π は，logical function symbol F を一つだけ含む term に対して用いられ，F が executable である場合，この Π literal を executable predicate とみなす。このため，spec では executable predicate list に，logical function を与えた。

LPS procedure は，この spec からつぎのように program を合成する。

Step 1.

conjecture の否定：1′. ¬P(x,y,z) に additional expression (a-exp) [(x,y,z),Φ] を付加する。(Φ は空集合)

Step 2.

{1′, 2, 3, 4, 5, 6} なる clauses の集合から，calculus K_1 によって □ (empty clause) の導出 (K_1-deduction) を行う。この導出を，conditional refutation (c-refutation) と呼ぶ。

図2.1に，3つの c-refutation を示した。各 tree の R と S の記号は，binary resolution と S-rule の適用を示す。

conjecture の否定，およびその子孫の clause を vital clause と呼ぶ。

axioms が無矛盾であれば，各 c-refutation によって得られる □ は，vital empty clause となる。

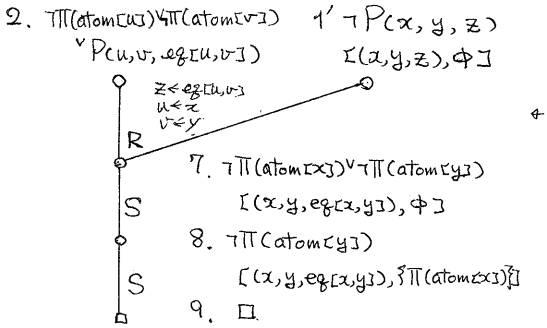
Step 3.

各 c-refutation の □ の a-exp を取り出し，下のような Recursive 定義系を作る。この一行を conditional term (c-term) と呼ぶ。また，Π literal から Π を取り除き，logical function を predicate に戻す。

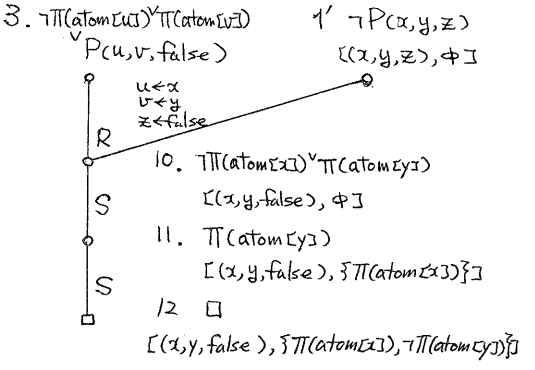
$$equal[x,y] = \begin{cases} eq[x,y] & \text{if } atom[x], atom[y] \quad (a)\text{より} \\ false & \text{if } atom[x], \neg atom[y] \quad (b)\text{より} \\ false & \text{if } \neg atom[x], atom[y] \\ and[equal[car[x], car[y]], & \\ \quad equal[cdr[x], cdr[y]]] & \\ & \text{if } \neg atom[x], \neg atom[y] \quad (c)\text{より} \end{cases}$$

図2.1. c-refutation trees of equal

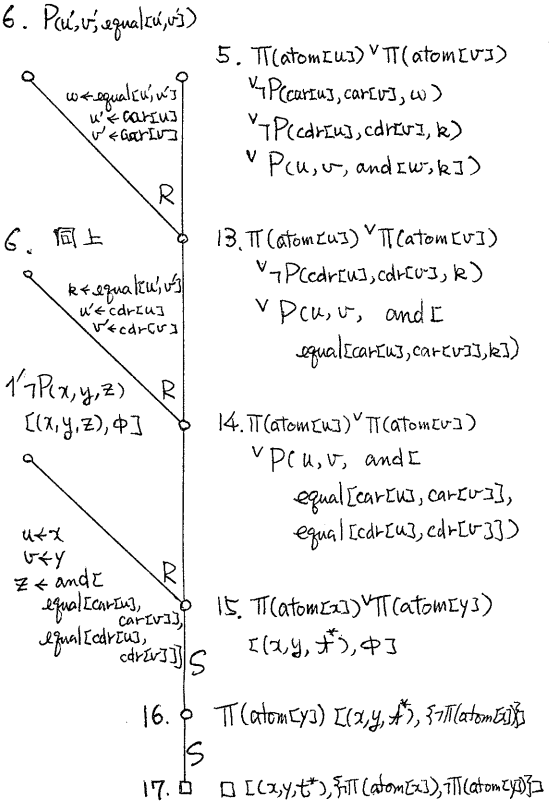
(a)



(b)



(c)



ここで、
$\alpha^* \equiv and[equal[car[x], car[y]], equal[cdr[x], cdr[y]]]$

さて、equal の spec において axiom 5 をつぎの2つの clause で置きかえてもよい。(付録例1参照)

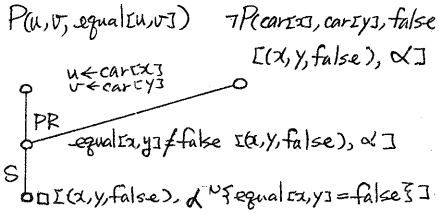
5.1. $\Pi(atom[u]) \vee \Pi(atom[v])$
$\vee \neg P(car[u], car[v], false) \vee P(u, v, false)$

5.2. $\Pi(atom[u]) \vee \Pi(atom[v]) \vee P(car[u], car[v], true)$
$\vee \neg P(cdr[u], cdr[v], w) \vee P(u, v, w)$

これは、$equal[car[u], car[v]]$ の値によって場合分けをしたものである。

この新しい spec からの c-refutation には、図2.2 のようなものが現れる。

図2.2. mgpu binary resolution for logical constant (1', 5.1 より)



ここに、$\alpha = \{\neg\Pi(atom[x]), \neg\Pi(atom[y])\}$
PR は mgpu binary resolution を示す。

これまでに用いられた LPS calculus の変形規則を示す。ここで、C, C_1, C' は clause、$\mathcal{F}, \mathcal{F}_1$ は term の並び、$L(\mathcal{F})$ は $\mathcal{F}$ の出現を含む literal、$\bar{L}$ は L の complementary literal、$\mathcal{F} \neq \mathcal{F}'$ はすべての $f_i \neq f_i'$ の disjunction からなる clause、α は literal の集合を表わす。

1. binary resolution.

$$\frac{C_1 \quad C_2[\mathcal{F}, \alpha]}{C[\mathcal{F}\sigma, \alpha\sigma]}\ \sigma$$

$C_1 : C_1' \vee L(\mathcal{F}_1)$
$C_2 : C_2' \vee \bar{L}(\mathcal{F}_2)$
$\sigma : \mathcal{F}_1\sigma \equiv \mathcal{F}_2\sigma$ なる mgu (most general unifier)

ただし、C_2 も non vital clause の場合は、a-exp はなくなる。

2. S-rule.

Δ を literal の集合とするとき、

$$\frac{C_1[\mathcal{F}_1, \alpha_1]}{C[\mathcal{F}_1, \alpha]}$$

$C_1 : C_1' \vee L$
$C : C_1'$
$\alpha : \alpha_1 \cup \{\bar{L}\}$
ただし、$L \in \Delta$

3. mgpu binary resolution

$$\frac{C_1 \quad C_2[\mathcal{F}, \alpha]}{\mathcal{F}_1 \neq \mathcal{F}_2' \vee C[\mathcal{F}\sigma, \alpha\sigma]}\ \sigma$$

$C_1 : C_1' \vee L(\mathcal{F}_1)$
$C_2 : C_2' \vee \bar{L}(\mathcal{F}_2)$
$C : C_1'\sigma \vee C_2'\sigma$

$(\sigma, \mathcal{F}_1 \neq \mathcal{F}_2)$: $[\mathcal{F}\sigma]^{\mathcal{F}_1}_{\mathcal{F}_2} \equiv [\mathcal{F}_2\sigma]^{\mathcal{F}_1}_{\mathcal{F}_2}$ なる most general partial unifier, および unification condition.

ただし、$[e]^{\mathcal{F}}_{\mathcal{F}'}$ は表現 e のすべての $\mathcal{F}$ を $\mathcal{F}'$ で replace した表現である。

これら以外に、vital clause 間の binary resolution, binary factoring などがある。

Ⅲ pilot1 の implementation.

1. factoring の適用法

つぎの例に見られるように，factoring が可能な clause(1)がある場合，TP(Theorem Prover)では，それを factor(イ)で置きかえ，イを含む(c-)refutation は求めない。しかし，LPS では clause4 のように，イからでは得られない a-exp が，イから求められるため，イを含む(c-)refutation も求めなくてはならない。

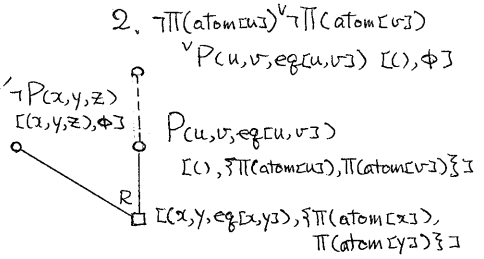
1. $P(x,y)\vee P(x,y)\vee R(z)$ $[(x,y,z),\{Q(x)\}]$
イ. $P(x,y)\vee R(z)$ $[(x,y,z),\{Q(x)\}]$
2. $\neg P(f(u),v)$
3. $\neg P(u,g(v))$
4. $R[z]$ $[(f(u),g(v),z),\{Q(f(u))\}]$ 1,2,3 より

2. S-rule の適用法.

pilot1 では，つぎの2つの理由により，non vital clause にも a-exp: $[(),\alpha]$ を加えて，S-rule を前処理として行う。これは，無駄な resolution を除き効率を向上させる。

S-rule によって a-exp に与される literal は，
i) それを resolved upon して得られる (c-) refutation が不要であること。
ii) K1 に基づく LPS procedure では，clause 内に復帰しないこと。

図3.1.



点線は前処理を表わす。

この non vital clause の a-exp に関して，vital clause 間以外の resolution の変形図は，つぎのようになる。

$$\frac{C_1[(),\alpha_1]\quad C_2[t_2,\alpha_2]}{C[t_2\sigma,\alpha_1\sigma\vee\alpha_2\sigma]}\ \sigma$$

記号は前述の変形規則と同様。

3. IDaxiom による前処理.

equal の spec の axiom5 は，program の recursion を表わし，自分自身と resolution を行い，無限に resolvents を作り出す。IDaxiom は，このような無限の resolvents を不要なものとする。

IDaxiom は，他の axiom と区別せずに用いた場合，それらの resolvents を禁止できない。この禁止を行うため，axioms と IDaxiom の resolution を前処理として行う。

図2.1(c)において，axiom5 は前処理により clause14 に置き換えられ，{1', 2, 3, 4, 14} から，c-refutation の構成を行うことになる。

pilot1 は，この前処理を単純な algorithm によっているため，完備(complete)な処理になってはいないが，無限の resolvents の生成を，多くの場合有限に押え，効率を向上させている。

spec が，互いに関係を持つ複数の program (subroutine など) に関する spec から構成されている場合，各 program に対して，conjecture が与えられるので，これを multi-conjecture と呼ぶ。

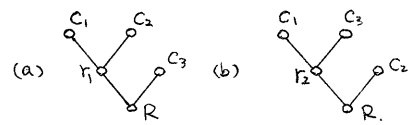
multi-conjecture の場合に，pilot1 は各々の conjecture とすべての axioms からなる部分 spec を用いて，各 program の合成を行う。

この場合の IDaxioms の前処理は，すべての conjecture に対する IDaxioms によって行う。これは，1つの conjecture に対する IDaxiom の前処理だけでは，他の conjecture に対する axioms によって発生する無限の resolvents の生成を禁止できないからである。(付録例2を参照)

4. clashによる処理.

図3.2のように，clause(1,2,3)からの異なるdeduction treeが，同一のresolvent(R)を持つ場合，clauseの集合$\{C_1,C_2,C_3\}$をclashと呼ぶ。

図3.2.



clashを用いて同一resolventの重複を減少させるために以下の方法を用いる。

まず，入力clausesを順序づけ，各resolventに対してどの入力clauseを何度用いたかを表わすindexを与える。たとえば，入力clauseの並びを(C_1,C_2,C_3)と定めると，図3.2のr_1のindexは$(1,1,0)$となる。

一方，一組のparent clauseから，複数のresolventsが得られる場合がある。これらのresolventsには同一のindexを対応させる。

つぎの手順により，clash checkを行う。はじめに，axiomsおよびconjecturesの否定からなる入力clauseにindexを対応させる。

step1：与えられたparent clauses C_1,C_2のindex α_1,α_2を求める。

step2：$\alpha_1+\alpha_2$(ベクトル和)なるindexがあるかどうか？ もしあれば，C_1,C_2のresolutionは行わない。もしなければstep3へ。

step3：C_1,C_2と同一のindexを持つclausesの集合F_1,F_2を求める。

step4：$F_1\times F_2$のすべての要素について，そのpairをparent clausesとするresolutionを行い，できたすべてのresolventsに同一のindex $\alpha_1+\alpha_2$を対応させる。

この方法では，同一resolventの重複を完全に禁止することはできないが，相当数の重複を禁止することができる。

5. 補助機能

デモンストレーションにおいて，作成されたprogramを実際に実行するために，得られたc-termをLisp programに翻訳する機能，問題や手法の評価のためにresolventやロの数を計測する機能，および(c)-refutationの構造の情報を作成する機能などを，pilot1は持つ。

翻訳機能では，specに不足があり，合成されたprogramに中間変数が現れる場合，ユーザーに介入を求める。

6 pilot1のsemi-algorithm

このsemi-algorithmは，breadth first methodに基づいたものである。

step1：axiomsに対する，IDaxiomsとS-ruleの前処理。

step2：conjectureを1つ設定。

step3：i)conjectureの否定にa-expを追加，ii)clash用table その他の初期値設定。

step4：$\lambda\leftarrow 1$
$S^\lambda\leftarrow$ axioms $\cup$ conjectureの否定。

step5：$\lambda\leftarrow\lambda+1$.

step6：clauseの集合$S^1\cup\cdots\cup S^{\lambda-1}$と$S^{\lambda-1}$とから，次のclauseのpair C_1,C_2を選び出す。全てのpairが選択された時，step10へ。

step7：C_1,C_2に対するclashのindexをcheckする。同一indexが登録済みであれば，step6へ。

step8：$[C_1]^{(1)},[C_2]$の直積の各pairに対してresolutionを行う。生成されたresolventsをS^λにセットする。ロが生成された場合はS^λにセットせず，そのa-expをc-termに変換する。

step9：step8で作られたresolventsに対応するindexの登録。step6へ。

step10：S^λがemptyでなければ，step5へ。

step11：未処理のconjectureがあれば，step2へ。それ以外は，c-termの並びを実行可能なprogramに翻訳する。

$[C_1]^{(1)}$：C_1と同一のclash indexを持つclausesの集合を示す。

なお，pilot1では，resolventが作られるたびに，その番号をターミナルに表示する。そして，ユーザーの判断により，INTERLISPの機能を用いて割り込みを行い，状況に応じて適当な処理を施すことができる。

図3.3にpilot1全体のフローを示した。

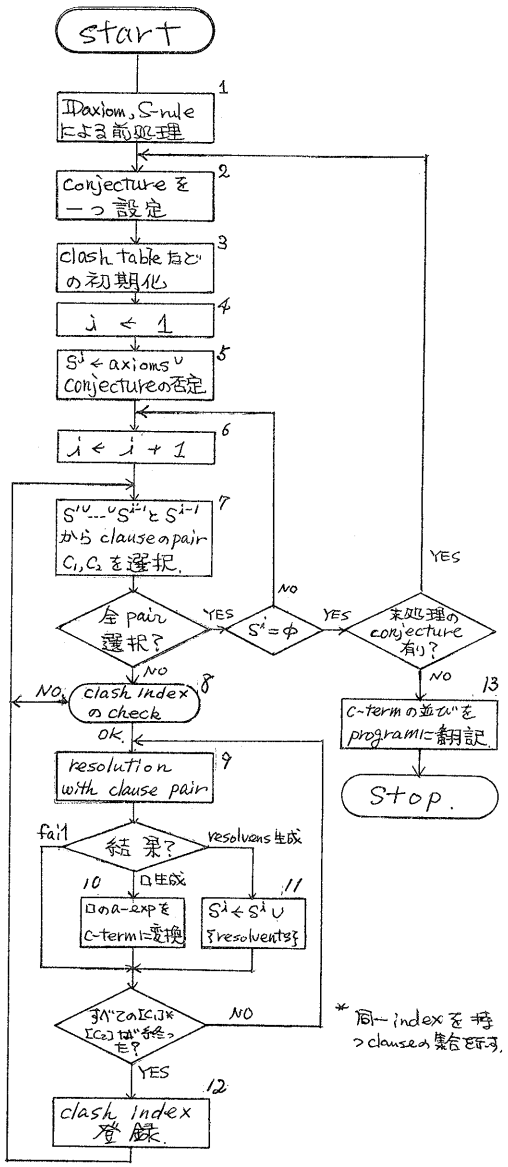


図3.3 pilot1のflow chart.

IV. pilot1のシステム構成

function $f(t_1,\cdots,t_n)$, predicate $P(t_1,\cdots,t_m)$ は，それぞれ，リスト表現 $(f\ t_1\cdots t_n)$, $(P\ t_1\cdots t_n)$ で，また，$\neg P(t_1,\cdots,t_m)$ は (NEG $P\ t_1\cdots t_m$) で表わす。

clauseは，(N ∀B よβα) なるリストで表わす。ここで，Nはclause番号，∀はB中に現れる変数のリスト，Bはclauseの本体でliteralの並び，よ,β,αはa-expに対応したtermの並びおよびliteralの並びで，(入力時には不要である)，ある。たとえば，3. $P(b,f(g(c)))\vee\neg R(y)$ の入力形式は，(3 (Y) ((P b (f (g c))) (NEG R Y))) となる。

historyは，(10 2 4 (1) (3)) のようなリスト表現の並びである。これは，clause 10がclause 2とclause 4のresolventであり，その最初のliteralと4の3番目のliteralとがresolved uponされたことを示す。このリストはhistと呼ぶ。

pilot1は，つぎの入力形式で呼ばれる。

LPS[pname, pspec, P, F, V, C, Pflag]

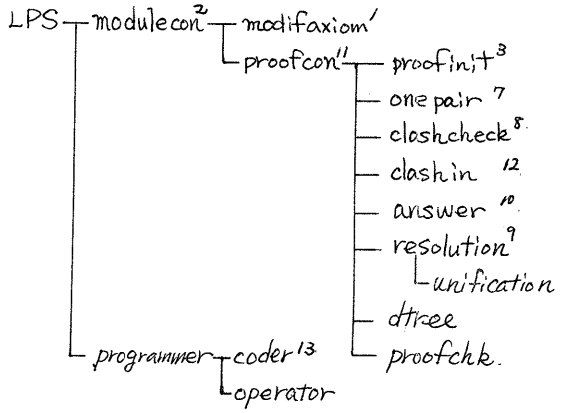
ここで，

pname は，specの title。
pspec は，conjectureの否定，axioms。
IDaxioms の並び，

Pは，spec 中に用いられるexecutable predicate symbol の並びである。

Fは，引数の数と組になった出力関数の並びである。
Vは，入力，中間，出力変数の並び。
Cは，logical constantsの並び。
Pflagは，print flagで，実行中の表示の要(T)，不要(NIL)を示す。

pilot1のprogram structureを示すと，つぎのようになる。(右肩の数字は図3.3のbox番号と対応)



Main program LPSは，modulecon と programmer とで構成されている。

moduleconは，IDaxiom と S-rule による前処理を modifaxiom で行い，spec が multi-conjecture の場合の，conjecture による制御をし，c-term の並びを出力する。

programmerは，moduleconで作られた c-term の並びを，coderによって実行可能な Lisp program に翻訳し，operatorによって，ユーザーのターミナルからの指示に従い，そのprogramをLispシステム上に DEFINE またはディスクファイルに登録する。

proofconは，1つの conjecture の否定と前処理された axioms から c-refutation をいくつも構成し，c-refutation から情報を取り出す process を制御する。また，resolvent など clause の管理も行う。

proofcon を構成する主要な routine は，proofinit，onepair，clashcheck，clashin，resolution，answer などである。

proofinitは，conjecture の否定に対して a-exp を加え，clash用の table などの初期値設定を行う。

onepairは，2つの clause の集合から，つぎつぎと新しい clause の pair を選び出す。すべての pair を選択した場合，その終了を示す。

clashcheckは，onepairによって選ばれた clause の pair の clash index により clash の check を行う。

clashinは，resolvents の clash index を登録する。

answerは，ロの a-exp を c-term に変形する。

resolutionは，resolution と factoring を行い，resolvent と共に hist を作り出す。また，a-exp に関する操作も行う。mgu を求める処理は，unificationが行う。

unification では，literals 間の mgu 計算と共に，logical constant に対する unification condition も求める。

これら以外に，(c-)refutation tree に対応した history を作る dtree，history から axioms 自体が矛盾しているかどうかを調べる proofchk がある。

V. おわりに

pilot1は，琴野，大村，宮沢の卒業論文として，DEC system-20[1] (TOPS-20[2]) の INTERLISP[3] サブシステム上で作成された。直接の作業期間は約一ヶ月，作成されたprogramの大きさは，Lisp pretty print 形式で約3000行である。

三人がLispの初心者であり，INTERLISPの入手が七月であったこと，およびpilot1への着手が九月末であり，前記研究会まで一ヶ月半弱と期間が限られていたことから，pilot1は TPU[5] の routine を基にしたLPS理論の直接的 implementとなり，programとしては洗練されたものではないが，そのわりには大規模なものとはならなかった。これは，LPS procedure が，極めてシンプルであることを，実証するものである。[4]

注. 1) Main Memory 256 KW (36 bit/w)，front-end processer として PDP-11 をもつ。
2) 仮想記憶256 kw を管理し，タイムシェアリングシステム，ファイルシステム等を持つ マルチモードオペレーティングシステム。
3) Xerox と BBN により，DEC system-10 上に共同開発された総合的な Lisp システム。全システムは 275 kw，常駐部は 152 kw の記憶容量を占める。
4) OL resolution に基づく次期実験システム(謝)は，coder を除いて約400行である。

また，初期の目的であるevalquote等の合成には，pilotは充分な機能と，デモンストレーションの限られた時間内での実行に充分耐える効率をもつ。

一例として，付録例3によって計算時間を測定した結果を，表5.1に掲げる。

表5.1 S-rule, clashの評価表.

	S-rule, clash 共になし	S-rule のみ	S-rule, clash 共に使用
生成resolvents数	storage full になり不明	16	12
生成(C-)refutation数	14コまで	17	6
clash checkで禁止された数	0	0	15
CPU time (real time)	不明 (約2時間で中断)	230 (973)	60 (260)

(単位 秒)

生成resolvent計数には口は含まない。

時間基準として，INTERLISPでは，1 consに0.00で秒要する。

ユーザーエリアに許されているfree spaceは，スタック用に13824W，それ以外に13824Wである。

programの実行は，インタープリター形式による。compileを行うと，8～10倍高速になる.

謝辞

日頃，御指導いただく，京都産業大学計算機科学研究所の上村義明教授に感謝します。

参考文献

1. 謝(1977)： Foundation of Logical Program Synthesis
ソフトウエア工学研究会資料 4-1.
2. 謝(1975)： プログラム シンセシスのための情報抽出系
信学会 AL 75-13.
3. 謝(1975)： Resolution Refutation 法による Mechanical Program Synthesis
信学会 AL 74-40
4. 謝(1975)： Mechanical Flowchart Synthesis について，
信学会 AL 75-2.
5. Chang & Lee (1973)： Symbolic Logic and Mechanical Theorem Prover, Academic Press.
6. Nilson, N.J. (1971)： Theorem Proving Methods in Artifical Intelligence, McGraw-Hill
7. Teitelman, W. (1974)： INTERLISP reference manual, Xerox
8. McCarthy, J. 他 (1962)： Lisp 1.5 programmer's manual The MIT press

付録

例 1

```
_PP SEQL

  (LPS (QUOTE (** S-EQUAL **))
       [QUOTE ([[(1 (X Y Z)
                    ((NEG EQUALP X Y Z]
                [[2 (I J)
                    ((NEG TT (ATOM I))
                     (NEG TT (ATOM J))
                     (EQUALP I J (EQ I J]
                 (3 (I J)
                    ((NEG TT (ATOM I))
                     (TT (ATOM J))
                     (EQUALP I J FALSE)))
                 (4 (I J)
                    ((TT (ATOM I))
                     (NEG TT (ATOM J))
                     (EQUALP I J FALSE)))
                 (5 (I J)
                    ((TT (ATOM I))
                     (TT (ATOM J))
                     (NEG EQUALP (CAR I)
                          (CAR J)
                          FALSE)
                     (EQUALP I J FALSE)))
                 (6 (I J K)
                    ((TT (ATOM I))
                     (TT (ATOM J))
                     (NEG EQUALP (CAR I)
                          (CAR J)
                          TRUE)
                     (NEG EQUALP (CDR I)
                          (CDR J)
                          K)
                     (EQUALP I J K]
                [((7 (I J)
                     ((EQUALP I J (EQUALF I J]
        (QUOTE ((TT)))
        [QUOTE (((EQUALF 2]
        (QUOTE ((X Y)
                (I J K)
                (Z)))
        (QUOTE ((TRUE FALSE)))
        (QUOTE T))
SEQL
_(TIME (EVAL SEQL) 1 0)

(1 CONTRADICTION 1 2)
((Z <- (EQ X Y)) if (TT (ATOM X)) (TT (ATOM Y)))

(2 CONTRADICTION 1 3)
((Z <- FALSE) if (TT (ATOM X)) (NEG TT (ATOM Y)))

(3 CONTRADICTION 1 4)
((Z <- FALSE) if (NEG TT (ATOM X)) (TT (ATOM Y)))

(4 CONTRADICTION 1 5)
((Z <- FALSE) if (NEG TT (ATOM X)) (NEG TT (ATOM Y)) ((
EQUALF (CAR X) (CAR Y)) = FALSE))

(5 CONTRADICTION 1 6)
((Z <- (EQUALF (CDR X) (CDR Y))) if (NEG TT (ATOM X))
(NEG TT (ATOM Y)) ((EQUALF (CAR X) (CAR Y)) = TRUE))

... Statistics Report need ?  : No


(EQUALF
  [LAMBDA (X Y)
    (PROG (Z)
          (if (ATOM X) and (ATOM Y)
              then (Z _ (EQ X Y))
                   (RETURN Z))
          (if (ATOM X) and (NOT (ATOM Y))
              then (Z _ NIL)
                   (RETURN Z))
          (if (NOT (ATOM X)) and (ATOM Y)
              then (Z _ NIL)
                   (RETURN Z))
          (if (NOT (ATOM X)) and (NOT (ATOM Y))
              and (NOT (EQUALF (CAR X)
                               (CAR Y)))
              then (Z _ NIL)
                   (RETURN Z))
          (if (NOT (ATOM X)) and (NOT (ATOM Y))
              and (EQUALF (CAR X)
                          (CAR Y))
              then (Z _(EQUALF (CDR X)
                               (CDR Y)))
                   (RETURN Z]))
5468 conses
12.492 seconds
92.487 seconds, real time
NIL
_
```

例 2

```
_PP SMULT

  (LPS (QUOTE (** MULTIPLICATION INCLUDE ADD **))
       [QUOTE ([[(1 (X Y Z)
                    ((NEG M X Y Z]
                 (2 (X Y Z)
                    ((NEG A X Y Z]
                [(3 (J)
                    ((M 0 J 0)))
                 (4 (I J K L)
                    ((NEG M I J K)
                     (NEG A J K L)
                     (M (ADD1 I)
                        J L)))
                 (5 (J)
                    ((A 0 J J)))
                 (6 (I J K)
                    ((NEG A I J K)
                     (A (ADD1 I)
                        J
                        (ADD1 K]
                [((7 (I J)
                     ((M I J (SMULT I J]
                  (8 (I J)
                     ((A I J (SADD I J]
       NIL
       [QUOTE (((SMULT 2)
                (SADD 2]
       (QUOTE ((X Y)
               (I J K L)
               (Z)))
       NIL
       (QUOTE T))
SMULT
_(TIME (EVAL SMULT) 1 0)

(1 CONTRADICTION 1 3)
((Z <- 0) if (X = 0))

(2 CONTRADICTION 1 4)
((Z <- (SADD Y (SMULT I3 Y))) if (X = (ADD1 I3)))

... Statistics Report need ?  : No

(1 CONTRADICTION 2 5)
((Z <- Y) if (X = 0))

(2 CONTRADICTION 2 6)
((Z <- (ADD1 (SADD I Y))) if (X = (ADD1 I)))

... Statistics Report need ?  : No

### SMULT ###
1       (if (X = 0) then (Z _ 0) (RETURN Z))
2       (if (I3 = (SUB1 X)) then (Z _ (SADD Y (SMULT (SUB1
X) Y))) (RETURN Z))

line number 2 has an unsatisfiable condition.
You may change them :

:> (if (I3 = (SUB1 X)) then (Z _ (SADD Y (SMULT (SUB1
X) Y))) (RETURN Z))
<> (C : (X GT 0))
<> P
:> (if (X GT 0) then (Z _ (SADD Y (SMULT (SUB1 X) Y)))
(RETURN Z))
<> OK

### SADD ###
1       (if (X = 0) then (Z _ Y) (RETURN Z))
2       (if (I = (SUB1 X)) then (Z _ (ADD1 (SADD (SUB1 X) Y)
))
(RETURN Z))

line number 2 has an unsatisfiable condition.
You may change them :

:> (if (I = (SUB1 X)) then (Z _ (ADD1 (SADD (SUB1 X) Y)))
(RETURN Z))
<> (C : (X GT 0))
<> P
:> (if (X GT 0) then (Z _ (ADD1 (SADD (SUB1 X) Y))) (RETURN
Z))
<> OK


(SMULT
  [LAMBDA (X Y)
    (PROG (Z)
          (if (X = 0)
              then (Z _ 0)
                   (RETURN Z))
          (if (X GT 0)
              then (Z _(SADD Y (SMULT (SUB1 X)
                                      Y)))
                   (RETURN Z))

(SADD
  [LAMBDA (X Y)
    (PROG (Z)
          (if (X = 0)
              then (Z _ Y)
                   (RETURN Z))
          (if (X GT 0)
              then (Z _(ADD1 (SADD (SUB1 X)
                                   Y)))
                   (RETURN Z))
3568 conses
14.668 seconds
278.95 seconds, real time
NIL
```

例 3

```
_PP SGRN

  (LPS (QUOTE (** GREEN **))
       (QUOTE ([((1 (X Z)
                    ((P X Z)))
                 (2 (Y W)
                    ((NEG P Y W]
                [[3 (I)
                    ((NEG P (QUOTE A)
                          I)
                     (P (QUOTE B)
                        (f I]
                 [4 (J)
                    ((NEG P (QUOTE B)
                          J)
                     (NEG q J)
                     (P (QUOTE C)
                        (g J]
                 (5 (K)
                    ((NEG P (QUOTE B)
                          K)
                     (q K)
                     (P (QUOTE C)
                        (h K]
                NIL))
       (QUOTE ((q)))
       [QUOTE (((SPATH 3]
       (QUOTE ((X Y Z)
               (I J K)
               (W)))
       NIL
       (QUOTE T))
SGRN
_(TIME (EVAL SGRN) 1 0)

(1 CONTRADICTION 2 1)
((W <- Z) if (Y = X))
6 7 8 9 10 11 12 13
(2 (9 1 3 (1) (1)) (CONTRADICTION 2 9))
((W <- (f Z)) if (X = (QUOTE A)) (Y = (QUOTE B)))

(3 (10 1 4 (1) (1)) (CONTRADICTION 2 10))
((W <- (g Z)) if (X = (QUOTE B)) (q Z) (Y = (QUOTE C)))

(4 (11 1 5 (1) (1)) (CONTRADICTION 2 11))
((W <- (h Z)) if (X = (QUOTE B)) (NEG q Z) (Y = (QUOTE
C)))
14 15 16 17
(5 (7 2 4 (1) (2)) (9 1 3 (1) (1)) (CONTRADICTION 7 9))
((W <- (g (f Z))) if (Y = (QUOTE C)) (X = (QUOTE A)) (q
(f Z)))

(6 (8 2 5 (1) (2)) (9 1 3 (1) (1)) (CONTRADICTION 8 9))
((W <- (h (f Z))) if (Y = (QUOTE C)) (X = (QUOTE A)) (NEG
q (f Z)))

... Statistics Report need ?  : Yes

<<<<<<    Statistics  Report    >>>>>>

 # of candidate pair      : 136
 # of clashed pair        : 15
 # of generated resolvent : 12
 # of kept resolvent      : 12
 # of success unification : 18
 # of empty clause(proof) : 6


(SPATH
  [LAMBDA (X Y Z)
    (PROG (W)
          (if (Y = X)
              then (W _ Z)
                   (RETURN W))
          (if (X =(QUOTE A)) and (Y =(QUOTE B))
              then (W _ (f Z))
                   (RETURN W))
          (if (q Z) and (X =(QUOTE B))
              and (Y =(QUOTE C))
              then (W _ (g Z))
                   (RETURN W))
          (if (NOT (q Z)) and (X =(QUOTE B))
              and (Y =(QUOTE C))
              then (W _ (h Z))
                   (RETURN W))
          (if (q (f Z)) and (Y =(QUOTE C))
              and (X =(QUOTE A))
              then (W _ (g (f Z)))
                   (RETURN W))
          (if (NOT (q (f Z))) and (Y =(QUOTE C))
              and (X =(QUOTE A))
              then (W _ (h (f Z)))
                   (RETURN W))
32585 conses
69.939 seconds
269.418 seconds, real time
NIL
_
```

例1,2で合成させたProgramの実行.

```
_EQUALF(ABC XYZ)
NIL
_EQUALF(KSU KSU)
T
_EQUALF(CONS (C O N S))
NIL
_EQUALF((X Y Z) XYZ)
NIL
_EQUALF((A B C) (A B B))
NIL
_EQUALF((A P P E N D) (A P P E N D))
T
_

_SADD(0 51)
51
_SADD(18 375)
393
_SMULT(8 9)
9
_SMULT(9 27)
243
_

_TRACE(SADD SMULT)
(SADD SMULT)
_SMULT(2 3)

SMULT:
X = 2
Y = 3

   SMULT:
   X = 1
   Y = 3

      SMULT:
      X = 0
      Y = 3
      SMULT =

      SADD:
      X = 2
      Y = 3

         SADD:
         X = 2
         Y = 2

            SADD:
            X = 1
            Y = 2

SADD:
X = 2
Y = 2
SADD = 4
               SADD = 1
            SADD = 2
         SADD = 3
   SMULT = 3

   SADD:
   X = 3
   Y = 3

      SADD:
      X = 2
      Y = 3

         SADD:
         X = 1
         Y = 3

            SADD:
            X = 2
            Y = 3
            SADD = 5
         SADD = 4
      SADD = 5
   SADD = 6
SMULT = 6
6
_
```