ICOT Technical Memorandum: TM-0715

TM-0715

CILツールボックス説明書

ICOT第二研究室　CILグループ

March, 1989

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

CIL ツールボックス説明書

ICOT 第二研究室
CIL グループ

平成元年 3 月 3 日

はじめに

本書は、CIL ツールボックスのインスト・ル方法、および、ツールの説明について記述している。CIL,SIMPOS,ESP に関しては既知であることを前提としている。これらに関する説明は、以下の文献を参考のこと。

CIL に関する説明

CIL 操作説明書

SIMPOS,ESP に関する説明

SIMPOS 操作説明書
SIMPOS プログラミング説明書
ESP 説明書

目次

第I 部

インストール編

第 1 章

物理構成

1.1 リリース物件

次の二つがある。

1. CIL ツールボックス説明書 (本書) ..... 一部
2. リリース・ディスク ..... 一枚

1.2 稼働環境

CIL ツールボックスは CIL 第 3.6 版上で使用する。

H/W: PSI-II

OS: SIMPOS V4.2

S/W: CIL V3.6

第 2 章

インストール

2.1 手順

以下に CIL ツールボックスのインストール手順を示す。

1. PSI を立ち上げる。
2. CIL 第 3.6 版を起動する。
3. ユニット tool を作る。
   (a) CIL のコマンドメニュからunit(日本語モードではユニット) を選択する。
   (b) サブメニュが現れるので、make(日本語モードでは作成する) を選択する。
   (c) ユニット名を聞いて来るので、toolと入力する。
   以下は、英語モードで作成した例である。例中の xxx は、ユーザ名である。

```
CIL>make_unit.

  Unit name>tool

  Making Unit(tool) ...
    Making directory(me:cil.tool) ... Made
    Making package(xxx_tool) ... Made
  Unit(tool) is made
```

4. ユニット tool をセレクトする。
   (a) CIL のコマンドメニュからunit(日本語モードではユニット) を選択する。
   (b) サブメニュが現れるので、select(日本語モードでは選択する) を選択する。
   (c) ユニット一覧メニュが現れるので、toolを選択する。
5. ツールプログラムをユニットにコピーする。
   (a) ファイル・マニピュレータを起動する。
   (b) utility のmountコマンドを使って、リリース・ディスクをユニット 0 にマウントする。
   (c) ディレクトリ・コマンドのcopy_textを使って、フロッピー内のファイルをすべて、ディレクトリme:cil.toolにコピーする。

```
copy_text>*.*
to>me:cil.tool>*.*

>fd0>algo_w.cil ==> icpsixxx>sys>...>cil.tool>algo_w.cil. [ok]
                    :
```

(d) utility のdismountコマンドを使って、リリース・ディスクをディスマウントする。

6. ツールをツールボックスに登録する。

(a) CIL のコマンドメニュからCIL(日本語モードでは処理系) を選択する。

(b) サブメニュが現れるので、state(日本語モードでは状態を変更する) を選択する。

(c) ステートメニュが現れるので、public_declaration(日本語モードではパブリック宣言の適用) の項目をon(日本語モードではする) にしたのち、do_itをクリックする。

(d) CIL のコマンドメニュからtoolbox(日本語モードではツールボックス) を選択する。

(e) サブメニュが現れるので、catalog(日本語モードでは登録する) を選択する。

(f) ユニット tool 内に5cでコピーしたツールプログラム一覧メニュが現れる。以下に示すツールプログラムをクリックし、DO_IT(日本語モードでは 実行する) を選択する。これにより、プログラムはツールボックスに登録される。

i. algo_w.cil
ii. bool.cil
iii. dfasyn.cil
iv. finite.cil
v. finite.dcl
vi. func.cil
vii. ml.cil
viii. ml.dcl
ix. partial.dcl
x. subgen.cil
xi. trans.cil

以下は、英語モードでの登録実行例である。

```
CIL>catalog_toolbox.

  Catalogging(finite.dcl.1) ... Completed 00:00:01
                          :
CIL>
```

2.2 確認

2.2.1 登録確認

1. CIL コマンドメニュからtoolbox(日本語モードではツールボックス) を選択する。

2. サブメニュが現れるので、browse(日本語モードでは一覧を見る) を選択する。

3. メニュが現れるので、6fで登録したツールプログラムがメニュ中にあることを確認する。

2.2.2 実行確認

ユニット tool には、5cでコピーしたツールプログラムの他に、以下の例題プログラムが含まれている。

1. dfasyn_test.cil

2. partial_test1.cil

3. partial_test2.cil

4. subgen_test.cil

5. trans_test.cil

これらのプログラムの内、 partial_test1.cil を使って実行確認する。

1. 例題プログラムをコンパイルする。

   (a) CIL コマンドメニュからprogram(日本語モードではプログラム) を選択する。

   (b) サブメニュが現れるので、compile(日本語モードではコンパイルする) を選択する。

   (c) すると、ユニット中のプログラム一覧メニュが現れるので、例題プログラムpartial_test1.cilをクリックし、DO_IT(日本語モードでは実行する) を選択する。これにより、例題プログラムがコンパイルされる。

```
CIL>compile_program.

  Compiling(partial_test1.cil.1) ...
    Including declaration(partial.dcl.1) ... Included
  Completed 00:00:42

CIL>
```

2. 例題プログラムをコンサルトする。

   (a) CIL コマンドメニュからprogram(日本語モードではプログラム) を選択する。

   (b) サブメニュが現れるので、consult(日本語モードではコンサルトする) を選択する。

   (c) すると、ユニット中のプログラム一覧メニュが現れるので、1cでコンパイルした例題のコンパイルドプログラムpartial_test1.cmpをクリックし、DO_IT(日本語モードでは実行する) を選択する。これにより、例題プログラムがコンサルトされる。

```
CIL>consult_program.

  Consult_on(partial_test1.cmp.1) ... Completed 00:00:00

CIL>
```

3. 例題プログラムを実行する。
   CIL のトップレベルから、述語discourse(I,T)を実行する。以下に実行結果を示す。

```
CIL>discourse(I,T).

  I => [soa(love,(jack,betty,loc(1)),yes),soa(love,(betty,jack,loc(2)),yes)]
  T => [A,2]
  yes

CIL>
```

第II 部

説明編

第3章

概要

CILグループでは、CIL第四版の言語使用実現に向けて制約解決機構の充実、型の導入について活動をしてきた。また、処理系の改良実験として、部分項の操作・単一化の高速化や、マクロ展開アルゴリズムの簡単化を行った。これらの調査・実験活動を通していくつかのプログラムを試作した。これらのプログラムは、実用的なものや、汎用的なものであると思われるので、ここでは、それらのプログラムのうちのいくつかをツールとしてまとめて提供する。これらのツールは、CIL V3.6で機能追加されたツールボックス機能を利用して提供している。ツールボックスの操作方法等はCIL説明書を参照のこと。

第 4 章

ツール群

4.1 有限領域を対象とした単一化の拡張

4.1.1 概要

Prologの基本機能である単一化を拡張し制約型論理言語を実現しようという研究の一つとしてDincbasの有限領域を対象とした単一化の拡張がある[1]。単一化を変数の値と領域を対象とすることで、従来は評価可能になって初めて評価されていた制約をより積極的に利用しようというもので、制約充足問題に有効とされている。

4.1.2 実現方法

実現の際のポイントとしては、以下のものが挙げられる。

領域付変数の導入

領域付変数は、以下のようなシンタックスを持つ構造体として実現されている。H_V は通常のエルブラン変数、領域 Domain は差分リストで実現している。領域が更新される毎に差分リストも更新される。エルブラン変数と領域の間には同期がとれており、実行中に領域がつの値に絞られるとエルブラン変数はその値に具体化され、逆にエルブラン変数が具体化されると領域はその値のみとなる。

H_V :: Domain

unification の拡張と unequality の定義

ユニフィケーションは、変数領域の継承や領域間での相互作用が生じるように拡張する。

1. エルブラン変数と領域付変数を単一化すると、エルブラン変数はその領域付変数の領域と同じ領域を持つ様な領域付変数となる。（領域の継承）
2. 定数と領域付変数を単一化すると
   (a) 定数が領域付変数の領域に含まれていれば、領域付変数の変数部（D_list の最後尾）に定数が代入され、成功する。
   (b) 定数が領域付変数の領域に含まれていなければ失敗する。
3. 領域付変数 (X::Dx,Y::Dy) 同士を単一化すると、それらの共通領域を D とし、
   (a) D が空ならば失敗する。
   (b) D={v}(要素数が 1) ならば X=Y=v で、成功する。
   (c) その他の場合は、X=Y とし、Dx と Dy を D に更新して、成功する。

非同一性のチェックは、以下の通りに行われる。

1. X が領域付変数 (X::Dx) で、Y が定数ならば Dx から Y を取り除いた領域を D とし、
   (a) D={v}ならば X=v として、成功する。
   (b) それ以外ならば、Dx を D に更新して、成功する。
2. X が定数で Y が領域付変数 (Y::Dy) の場合は、それぞれを置き換えて1と同様
3. X と Y が定数の場合は、そのまま X\ =Y を実行する。

forward checking と look ahead による領域更新時のフィードバックメカニズム

拡張単一化や非同一性チェックによって関係付けられた変数同士は、まだ値が具体化していなくても、それぞれの領域情報を用いることによって、互いの領域を狭めることができる (forward checking)。

制約が四則演算から成る計算式などで与えられる場合、拡張単一化や非同一性チェックに対する forward checking で得られた「演算結果に対する新しい領域」を、計算式にフィードバックさせて、式中の変数の領域を狭めることが可能な場合がある (look ahead)。

更に、look ahead の結果が再び forward checking を呼び出すという具合に両者は循環し、お互いにフィードバックがかかり、変数の領域を更新して行く。

4.1.3 使用方法

このツールは、プログラムファイルと宣言ファイルの二つからなっている。プログラムファイルには拡張単一化のアルゴリズムが記述されている。宣言ファイルには述語の呼び出しを容易にするためにCILシンタックスシュガーの定義が記述されている。ツールを利用するには、これらのファイルをインクルードする。

1. ファイル

   プログラムファイル: finite.cil

   宣言ファイル: finite.dcl

2. 述語

   d_variable(H_V,Dom,D_V): (シンタックスシュガーでは D_V< −H_V~Dom)
   エルブラン変数 H_V に領域 Dom を割り付けた領域付変数 D_V を返す。領域 Dom は領域の値として許されるすべての値をリストで与えなければならない。

   d_variable_unequal(D_X,D_Y): (シンタックスシュガーでは D_X~=D_Y)
   領域付変数の非同一性のチェックを行う。

   d_variable_unify(D_X,D_Y): (シンタックスシュガーでは D_X=D_Y)
   領域付変数を拡張単一化する。

   adder(D_X,D_Y,D_Z): (シンタックスシュガーでは D_X+D_Y)
   領域付変数同志の加算を行う。

   multiplier(D_X,D_Y,D_Z): (シンタックスシュガーでは D_X*D_Y)
   領域付変数同志の乗算を行う。

4.1.4 使用例

以下に簡単な例を示す。変数 X,Y,Z に領域を割り付けて、加算を行った時に値が収束する例題を挙げる。

1. プログラム

```
% ツールのインクルード宣言
:-include_declaration(toolbox/finite_operator).
:-include_program(toolbox/finite_basic).
test(X,Y,Z):-
    XD <- X~[1,2,3], % 領域の割り付け
    YD <- Y~[2,3,4], % 領域の割り付け
    ZD <- Z~[1,2,3], % 領域の割り付け
    ZD = XD + YD.
```

2. 実行結果

```
CIL>test(X,Y,Z).
  X => 1
  Y => 2
  Z => 3
  yes
```

4.2 ブール環における単一化

4.2.1 概要

Dinebas は[1] において、ブール環と有限領域を計算領域とする意味単一化 (Semantic Unification) について述べている。意味単一化とは、形の上では等しくないがある解釈の基では等しいような 2 つの項を、意味的に等価であるとする単一化の枠組みである。例えば 1+2 と 3 や 1+X と 2 は、通常の単一化 (Prolog の単一化) においては項の形が異なるため単一化に失敗するが、自然数とその加算という解釈の基では単一化に成功し X=2 が得られる。特に、現版 CIL における制約記述述語（制約解消系）の機能強化の観点から、ブール環における意味単一化 (Boolean Unification: ブール単一化) に注目し、その実験プログラムを試作した。

4.2.2 実現方法

以下に、ブール単一化のアルゴリズムとその解説を示す。また、このアルゴリズムではブール項で表現された制約の（排他的論理和と積から成る）標準形と入力された項を標準形に変換する書き換えアルゴリズムについては触れていないため、項書き換えルーチンは別途用意した。

単一化アルゴリズム:Unification Algorithm

```
proc unify(var x,y : term)
{
 solve(exor(x,y))
}
proc solve(var t : term)
var t1,t2 : term;
var v,w : term;
{
 if t = 0
 then   return(succeed)
 else   if t = exor(and(v,t1),t2))
                % v is a variable and t1, t2 do not contain v
        then    {
                 if solve(and(exor(t1,1),t2)) = succeed
                 then   {
                         bind v to exor(and(exor(t1,1),w),t2)
                        % w is a new variable
                         return(succeed)
                        }
                 else   return(fail)
                }
        else    return(fail)
}
```

解説

入力である論理項 x,y は排他的論理和 (exor) と積 (and) から成るブール項 (Boolean Term) である。また、ステップ 2 は上記アルゴリズムには明記されていない補足部分である。

ステップ 1: $t = exor(x, y)$とする。

ステップ 2: tを排他的論理和と積に関する書き換え規則に従って、標準形 $t = exor(s_1, s_2, \ldots, s_n)$の形へ変換する。

ステップ 3: $t = 0$ならば単一化に成功して終了。
さもなくば、ステップ 4 へ。

ステップ 4: tを$exor(and(v,t_1),t_2)$の形に変換する。 但し、vは変数で、t_1とt_2はこれを含まない。 tが変数vを含んでいる場合、tの構成要素$s_1,s_2,\ldots,s_n$ をvを含んでいるものといないものとに分類し、前者のグループ からvを取り除いた結果の排他的論理和をt_1とし、後者のグル ープの排他的論理和をt_2とする。
変換に成功すれば、ステップ 5 へ。 さもなくば、失敗して終了。

ステップ 5: $t=and(exor(t_1,1),t_2)$としてステップ 2 へ
その結果が成功であれば、vに$exor(and(exor(t_1,1),w),t_2)$ を代入し、単一化に成功して終了。但し、wはこれまでに出現してい ない新しい変数。
結果が失敗であれば、失敗して終了。

問題点

このアルゴリズムは、従来は評価可能になって (制約内の変数に具体値が代入されて) 初めて評価されていた制約をより積極的に利用しようというアプローチの一つであり、その意味では能動的制約解消系と言える。
しかし、このアルゴリズムに対し次のような欠点も指摘されている[3]。

- ブール項で表現された制約の標準形を厳密に規定していないため、制約解消系の実装方法によっては、意味的には同じだが形の異なる制約 (ブール項) が出力として得られる場合がある。
- プログラムやゴールに含まれる以外の変数を、制約解消系が実行中に勝手に導入するため、出力として得られる制約 (ブール項) が複雑になる。

また、この制約解消系を CIL に導入するにあたっては、

- 処理全体の高速化と出力結果の簡潔化のための強力な項書き換えアルゴリズム
- 従来の単一化とブール単一化との融合

等の課題が残されている。

4.2.3 使用方法

このツールは、プログラムファイルをコンサルトするか、または、ユーザプログラム中にインクルードすることによって利用することができる。

1. ファイル

   ソースプログラム: bool.cil

2. 述語

   b_unify/2: ブール単一化述語
   第 1 引数のブール項と第 2 引数のブール項とが単一化可能か否かを判定する。もし可能ならばその際の unifier(変数への代入) も求める。内部では、第 1 引数と第 2 引数の排他的論理和が 0(0orl の 0) と成るか否かで判定を行っている。

4.2.4 使用例

例として、ツールボックスからツールをコンサルトして幾つかのブール単一化を行った実行結果を示す。

```
CIL>consult_toolbox.
  consult_on(bool.cmp) ... Completed 00:00:00

CIL>b_unify(A,B).            % 通常の単一化
  A => A
  B => A
  yes

CIL>b_unify(not(A),A).       % A と not A のブール単一化。
```

```
no

CIL>b_unify(and(A,B), or(A,B)).      % A かつ B と A または B の単一化
  A => A                             % 成功するのはAとBが同値の時である。
  B => A
  yes

CIL>b_unify(not(and(A,B)),or(not(A),not(B))).    % ド・モルガンの法則
  A => A
  B => B
  yes

CIL>b_unify(or(and(X,Y),not(and(X,Y))),Z).  % 恒真
  X => X                                    % (X and Y) or not(X and Y)
  Y => Y
  Z => 1
  yes
```

4.3 アルゴリズム W

4.3.1 概要

型推論が自然言語処理の立場からも注目されている[5]。ここで型推論とは、型の割り当てられていない対象に他の型情報に基づいて型を割り振るための推論過程のことをいう。例えば、aの型がαで関数$f(x)$の型が$\alpha \rightarrow \beta$であれば、$f(a)$の型はβであることが推論できる。ここでは、いろいろある型推論システムの中で型推論アルゴリズムがよく知られている ML をとりあげた。ML の型推論には多相型が導入されており、また型推論が自動化されている。ML の型推論を調査するために、その型推論アルゴリズムであるアルゴリズム W[4] をインプリメントした。

4.3.2 実現方法

アルゴリズムの概要

理論的な詳細は[4] に述べられている。ここでは[6] を参考に、基本用語の解説とアルゴリズムの概要説明を行う。

1. 型推論
   ここでは、型推論とは次のような意味である。識別子に関するいくつかの型宣言の集合Γが与えられたとき、まだ型の割りつけられていない表現eの型をΓ に基づいて推論すること。 表現eが型τに属することは$e : \tau$と書く。これを型宣言という。

2. 表現 (expression)
   ここでは、表現として次のものを扱う。識別子の集合を仮定し、それを基にして表現を定義する。

   <表現> ::= <識別子>
   | <表現> <表現>
   | λ. <識別子> <表現>
   | *fix*. <識別子> <表現>
   | *let* <識別子>=<表現> *in* <表現>
   | *if* <表現> *then* <表現> *else* <表現>

   識別子x、表現e, e', e''に対して、ee'はeへのe'の適用、 $\lambda x.e$はeからx の抽象、$fix\ f.e$は$\lambda f.e$の最小不動点である。 $let\ x = e\ in\ e'$はe'中のxをeで置き換えることを表わしている。 $if\ e\ then\ e'\ else\ e''$は通常の条件文である。

3. 型
   ここでは、型として次のものを扱う。基本型の集合と型変数の集合を仮定し、型を定義する。

   <型> ::= <基本型> | <型変数> | <構成型>
   <構成型> ::= <型> × <型> | <型> → <型> | <型> list
   <基本型> ::= int | bool | token | ...
   <型変数> ::= α | β | ...
   <型スキーム> ::= <型> | ∀ <型変数>. <型スキーム>

   型スキームは、全称束縛されている型変数に対する任意の代入による具体例の集合を表わす。 型1×型2は型1と型2の直積の型を、型1→型2は型1から型2への関数の型を、型1 list は型1の list の型を表す。

4. 推論規則
   ここでは、次のような型スキームに対して次のような推論規則を扱う。型宣言の集合 Γは一つの識別子については、高々一つしか仮定を含まない。$\Gamma[-x_1, \ldots, -x_n]$は、$\Gamma$から識別子$x_1, \ldots, x_n$ の型宣言を除去したもの。σ, σ'は型スキームを表わす。 τ, τ'は型を表わす。

   TAUT:

$$\frac{}{\Gamma[-x] \cup \{x : \sigma\} \vdash x : \sigma}$$

INST:

$$\frac{\vdash e : \sigma}{\Gamma \vdash e : \sigma'}$$

(σ'はσのインスタンスである)

GEN:

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha.\ \sigma}$$

(αはΓで自由でない)

COMB:

$$\frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e : \forall \alpha.\ \sigma}$$

ABS:

$$\frac{\Gamma[-x] \cup \{x : \tau'\} \vdash e : \tau}{\Gamma \vdash (\lambda x.\ e) : \tau' \to \tau}$$

FIX:

$$\frac{\Gamma[-f, -x] \cup \{x : \sigma\} \vdash e : \tau}{\Gamma \vdash (fix\ f.\ \lambda x.\ e) : \tau' \to \tau}$$

LET:

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma[-x] \cup \{x : \sigma\} \vdash e' \ \vdash \tau}{\Gamma \vdash (let\ x = e\ in\ e') : \tau}$$

COND:

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (if\ e_1\ then\ e_2\ else\ e_3) : \tau}$$

図 アルゴリズムの推論規則

5. アルゴリズム W

型推論アルゴリズム W は、型宣言の集合Γと表現eが与えられたとき、$\Gamma\theta \vdash e : \tau$となる主要型スキーム$\tau$と代入$\theta$を返す。主要型スキームとは、$\Gamma \vdash e : \tau$を満たす任意の$\tau'$に対して、$\Gamma\theta \vdash e : \tau$かつ$\tau' \leq \tau$を満たす型スキーム$\tau$を主要型スキームという。$\tau' \leq \tau$は、$\tau = \forall\alpha_1, \ldots, \forall\alpha_n.\ \sigma$でかつ$\beta_1, \ldots, \beta_n$を$\tau$の自由型変数でない変数とするとき、ある代入$\theta$が存在して$\tau' = \forall\beta_1, \ldots, \beta_m.\ \sigma\theta$が成立することを示す。

(a) $W(\Gamma, x) = ([\,], \tau\eta)$
　但し、$x : \forall\alpha_1 \ldots \forall\alpha_n.\ \tau \in \Gamma$
　τのすべての総称的型変数を新しい型変数に置換する代入をηとする。

(b) $W(\Gamma, e_1 e_2) = (\theta_1\theta_2\eta, \beta\eta)$
　但し、$W(\Gamma, e_1) = (\theta_1, \tau_1)$
　$W(\Gamma\theta_1, e_2) = (\theta_2, \tau_2)$
　$match(\tau_1\theta_2, \tau_2 \to \beta)$
　βは新しい型変数

(c) $W(\Gamma, \lambda x.e_1) = (\theta_1, \beta\theta_1 \to \tau_1)$
　但し、$W(\Gamma[-x] \cup \{x : \beta\}, e_1) = (\theta_1, \tau_1)$
　βは新しい型変数

(d) $W(\Gamma, fix f.\ e) = (\theta_1\eta, \tau_1\eta)$
　但し、$W(\Gamma[-f] \cup \{f : \beta\}, e) = (\theta_1, \tau_1)$
　$match(\beta\theta_1, \tau_1) = \eta$

(c) $W(\Gamma, let\ x = e_1\ in\ e_2) = (\theta_1\theta_2, \tau_2)$
但し、$W(\Gamma, e_1) = (\theta_1, \tau_1)$
$W(\Gamma[-x]\theta_1 \cup \{x : closure(\Gamma\theta_1)(\tau_1)\}) = (\theta_2, \tau_2)$

(f) $W(\Gamma, if\ e_1\ then\ e_2\ else\ e_3) = (\theta_1\eta_1\theta_2\theta_3\eta_2, \tau_2\eta_2)$
但し、$W(W, e_1) = (\theta_1, \tau_1)$
$match(\tau_1, bool) = \eta_1$
$W(\Gamma\theta_1\eta_1, e_2) = (\theta_2, \tau_2)$
$W(\Gamma\theta_1\eta_1\theta_2, e_3) = (\theta_3, \tau_3)$
$match(\tau_2, \tau_3) = \eta_2$

- $closure(\Gamma)(\tau)$
Γには出現しないτの自由型変数$\alpha_1, \ldots, \alpha_n$をすべて全称束縛した型スキーム。
つまり、$\forall\alpha_1, \ldots, \forall\alpha_n.\ \tau$
- $match(\sigma, \tau)$
二つの型スキームσ, τに対して、$\sigma\theta = \tau\theta$なる代入 θがあるとき最も一般的な代入θ'を求める関数。

4.3.3 使用方法

このツールは、プログラムファイルをコンサルトするか、または、ユーザプログラム中にインクルードすることによって利用することができる。

1. ファイル

ソースプログラム: algo_w.cil

2. 述語

typing/4: W のタイプチェッカー
上記のアルゴリズム W における、$W(\Gamma, e) = (\theta, \tau)$が typing$(e, \Gamma, \theta, \tau)$ に対応する。
Γとeは入力であり、θとτは出力である。

4.3.4 使用例

以下に、ツールボックスからツールをコンサルトして、幾つかの例を実行した結果を示す。

```
CIL>consult_toolbox.
  Consult_on(algo_w.cmp.1) ... Completed 00:00:00
```

- 環境が空の時、$\lambda x.\ x$の型は$\forall\alpha.\ \alpha \to \alpha$であることを示す。

```
CIL>typing((lambda, x, x), [], S, T).
  S = []
  T = all,A,fun,A,A
```

- $succ$という関数の型は整数から整数への関数であるという環境において、$\lambda x.\ succ\,x$(xを$succ$に適用した結果得られる表現を変数xで抽象した表現) の型は整数から整数への関数の型であることを示す。

```
CIL>typing((lambda, x, (app, succ, x)), [(succ, (fun,int,int))], S, T).
  S= [(A,int),(B,int)]
  T= fun,int,int
```

- 環境が空の時、表現$let\ i = \lambda x.\ x\ in\ i$の型は$\forall\alpha.\ \alpha \to \alpha$であることを示す。

```
CIL>typing((let,i,(lambda,x,x),(app,i,i)),[], S, T)
  S= [(A,fun,B,B),(C,fun,B,B)]
  T= all,B,fun,B,B
```

4.4 CIL トランスレータの簡単化技法

4.4.1 概要

従来、処理の類似した六つのパスから成っていた DEC10 Prolog 版 CIL のトランスレータを、マクロ定義を書き換え規則としてテーブル化するのとにより簡単にした。この時使用したテクニックの基本的アイデアについて紹介する。

4.4.2 実現方法

実現する際の基本的なアイデアは極めて簡単なもので、

1. サブタームを（バックトラックで）次々に生成する。

2. 指定パターンと同じサブタームを指定された項で置き換える（代入）。

という、AI プログラミングの分野においては良く知られている、サブタームの生成と代入の二つが中心的な部分である。具体的には、1の目的で subterm、 2の目的で subst というルーチンを導入し、これらを使って rewriting と narrowing のルーチンを構成している。変換の手続きは、およそ次の様である。トランスレータは、与えられたタームのサブタームを順々に生成し、定義されている変換テーブルを参照し、合致するパターンの部分を書き換えて行く。今回工夫した点は、従来のローカルな書き換えしか許さない関数型マクロ他に、ローカルなパターンに注目しつつ、ターム全体を書き換えられるような新しいマクロを導入したことである。このために、書き換えテーブルに全体を書き換えるための情報を追加した。これによって、例えば、遅延実行制御の CIL シンタックスシュガー print(?(X)) を freeze(X,print(X)) へ変換するのも容易で、ルール形式も自然に与えることができた。ローカルな書き換えしか許さない、ESP や CIL 第三版で提供されているマクロ機能では、不可能ではないにしても、ルールが繁雑になることは避けられないと思われる。

4.4.3 使用方法

このツールは、以下に示すファイルをツールボックスのコンサルト機能でコンサルトして使うか、または、インクルードして使うことができる。

1. ファイル

   ソースプログラム: trans. cil

   例題ファイル: trans_test. cil

2. 述語

   trans(X,Y): 展開述語
   CIL シンタックスシュガーの展開述語で、入力 X 中の CIL シンタックスシュガーを展開した形を Y に返す。

   macro(X,Y): 定義述語
   CIL シンタックスシュガーの定義述語で、パターン X を Y に書き換えるための定義をする。

4.4.4 使用例

使用例として例題ファイルの実行例を示す。例題ファイル中のプログラムは、述語 trans を使用して、展開前、期待される展開結果、実際の展開結果を表示するプログラムである。

上記の例題ファイルをコンサルトする。コンサルトが完了したら、ツールボックスからツールをコンサルトする。述語 test/0 を実行する。

```
CIL>consult_program.
  Consult_on(trans_test.cil.1) ... Completed 00:00:01
CIL>consult_toolbox.
  Consult_on(trans.cmp.1) ... Completed 00:00:01
```

```
CIL>test.
Source   => p(A?)
Expected => freeze(A,p(A))
Expanded => freeze(A,p(A))

Source   => p(A:f(A))
Expected => f(A),p(A)
Expanded => f(A),p(A)

Source   => p(A!a)
Expected => role(a,A,B),p(B)
Expanded => role(a,A,B),p(B)

Source   => p(@f)
Expected => freeze(A,f(A)),p(A)
Expanded => freeze(A,f(A)),p(A)

Source   => p({a/1,b/2})
Expected => mk_assoc((a/1,b/2),A),p(A)
Expanded => mk_assoc((a/1,b/2),A),p(A)
yes
```

4.5 決定性有限オートマトン合成器

4.5.1 概要

このプログラムは、正規表現を与えると決定性有限オートマトンを合成するプロトタイプ版である。このプログラムで使った手法は教科書にも記されているごく一般的な方法ではあるが、マクロ展開処理など応用範囲は広いと思われる。

4.5.2 実現方法

オートマトンモデルは普通、初期状態、最終状態、状態推移関数の集合の三つ組で指定することができる。プログラム中では、この三つ組は三つの述語によって指定するようにした。述語名は、利用者が自由に決められるようになっている。例として、初期状態を initial、最終状態を final、状態推移関数を next で表す場合を考える。

- 質問 inital(X) は、X に初期状態を返す。
- 質問 final(X) は、X が最終状態かどうか調べる。
- 質問 next(X,Y,Z) は状態 X に入力 Y が与えられた時の推移状態 Z を返す。

プログラム内では上のような述語がアサートされた形で作られ、オートマトンモデルの三つ組と対応する。

正規表現は普通、空系列、空集合、和、積、閉包およびアルファベットで表現される。プログラム中ではこれらのシンボルとの対応は自然にとられている。すなわち、あらかじめ意味付けされている物としては (epsilon 空系列)、empty(空集合)、+ (和)、^ (積)、* (閉包) で、その他の複合項はアトミックな入力用アルファベットシンボルとして利用することができる。

4.5.3 使用方法

このツールは、以下に示すファイルをツールボックスのコンサルト機能でコンサルトして使うか、または、インクルードして使うことができる。

1. ファイル

   ソースプログラム: dfasyn.cil

   例題ファイル: dfasyn_test.cil

2. 述語

   synam(N,I,F,T): 合成述語
   正規表現 N からその表現に応じたオートマトンを合成し、初期状態、最終状態、状態推移関数に対応する三つの述語名 I,F,T がアサートされる[1]。

   accepted(S,I,F,T): 検査述語
   入力文字列 S が正規集合、すなわち、synam で合成されたオートマトンに受理されるかどうかを調べる。

4.5.4 使用例

使用例として例題ファイルの実行例を示す。この例題は、二進数列を受理するオートマトンを合成し、二進数列 101.101 が受理されるかどうか確認するプログラムである。

上記の例題ファイルをコンサルトする。つぎに、ツールをツールボックスからコンサルトする。test/0 を実行する。

```
CIL>test.

synthesizing automaton for binary digis...end
checking input string "101.101" ... ok
```

[1] オートマトンを合成する際に、空集合の正規表現、つまり、synam(empty,...) を実行すると、いかなる入力をも受理しないオートマトンを合成するので注意が必要である。

```
yes
```

4.6 サブターム生成器

4.6.1 概要

あるタームが与えられた時、そのタームのサブタームを生成し，それぞれに対してなんらかの処理をするような場合がある。このような場合、何らかのフィルター機能を使用してタームに対する総当たり的な探索を避ける方法がある。このプログラムの基本的なアイデアは、サブタームを生成する際に、生成を許容するパスを正規表現で記述し、探索空間に制限を加え効率的な生成を行おうというものである。

4.6.2 実現方法

サブターム生成器とオートマトンの間のインターフェースは以下の四つの述語によって実現される。initial は初期状態、final は最終状態、next は状態推移関数に対応し、また、入力シンボルとシンボルタイプを symboltype で定義する。

- final(<オートマトン識別子>,<最終状態>)
- initial(<オートマトン識別子>,<初期状態>)
- next(<オートマトン識別子>,<状態>,<入力>,<推移状態>)
- symboltype(<シンボル>,<シンボルタイプ>)

initial、final、next は自動的にアサートされるが、symboltype はユーザが定義しなければならない。
タームのサブタームを特定するためのパスは、次のような一連の形式で表現される。

$$(f_1, arg_1), (f_2, arg_2), \ldots, (f_n, arg_n)$$

ここで、f_iはファンクタ、arg_iはアーギュメントの位置を示す。(f_i, arg_i)の形で示されるペアが事実上の入力シンボルとみなされる。このパスの形式のファンクタとして許されるものを指定することによって、サブターム生成のフィルタリングを行っている。例えば、ファンクタとして ',’/2 を許すような場合で、(a,b,c) のサブターム c を表すパスは (',’/2,2),(',’/2,2) と表す。

4.6.3 使用方法

このツールは、以下に示すファイルをツールボックスのコンサルト機能でコンサルトして使うか、または、インクルードして使うことができる。

1. ファイル

   ソースプログラム: subgen. cil

   例題ファイル: subgen_test. cil

2. 述語

   defam(A,N): オートマトン合成
   許容パスの正規表現 N をオートマトンに変換する。合成されたオートマトンは、識別子 A によって区別される。

   subterm(T,P,S,A): サブターム生成
   正規表現から合成されたオートマトン A に従って、ターム T のサブタームを生成して S に返す。生成は、バックトラックベースで行われる。P は S の T におけるアーギュメントの位置を返す。

   symboltype(S,T): シンボルタイプ定義
   シンボル S、シンボルタイプ T を定義する。

4.6.4 使用例

使用例として例題ファイルの実行例を示す。これは、ターム (a;[b,(c,d)]) のサブタームを生成する例である。上記の例題ファイルをコンサルトする。test_def/0 を実行してから、引き数を変数のままで test/1 を実行する。redo すると順番に a,b,c,d,[] の値がバインドされる。

```
CIL>consult_program.
  consult_on(subgen_test.cil.1) ... Completed 00:00:00
CIL>consult_toolbox.
  consult_on(subgen.cmp.1) ... Completed 00:00:01
CIL>test_def.
Remember to definie <symboltype predicates.
yes
CIL>test(S).
S=a;
S=b;
S=c;
S=d;
S=[];
no
```

4.7 部分項の高速化マクロ

4.7.1 概要

CIL のユーザーシンタックスシュガーを使用して、部分項をすべての属性をもったエルブラン項に変換し、KL0 レベルで単一化できるようにした。プログラム中に出現する部分項の属性をすべて集めることでも、エルブラン項の次数（大きさ）を知ることが出来るが、今回はプログラム記述者に型宣言させる方法を採った。これは、

- 記述能力拡張の一つとして、CIL 第3.0 版で LOGIN[8] 風の継承機能を取り込んだ際に導入した型宣言（部分項の名前つけ）が利用できる。
- その際の宣言はプログラムの静的検査にもなる。

という理由による。

さらに、継承情報がエルブラン項を使った1次元の情報に変換でき、KL0 の単一化の枠組みで実現可能であることがわかったので、部分的にではあるが継承機能の導入も行った。なお、ここでいう継承関係は，次の三つの機能を想定している。

- 単一化可能性をチェックする。すなわち上下関係にない型のデータは単一化で失敗させる。
- 下位型との単一化に伴って絞り込まれたものも含め、データの型の計算を行う。たとえば、man 型の部分項は，その下位型である student 型の部分項と単一化すると、student 型になる。
- 多重継承に関しても同様の機能を許す。

時間的な性能に関しては、部分項が利用されている 40 行程度のプログラムを用いて実行時間を比較したところ、5~7 倍程の性能向上が得られた。

4.7.2 実現方法

今回の高速化は CIL 第三版のユーザーシンタックスシュガーを利用して実現している。

ラベル情報の収集と変換結果

型宣言全体を CIL 単一化することで、各型に割り当てるべきエルブラン項の大きさを決める。再帰 (無限) 型を含むような型宣言でも、CIL の単一化を用いればその型に対するエルブラン項の大きさを知ることが出来る。また型宣言の矛盾のチェックや、併合などが全て CIL の単一化の枠組みで実現可能である。例えば、

```
begin_definition.
    man <= {name/_, birthday/date, father/man}.
    date <= {year/_, month/_, day/_}.
    man <= {address/_}.
end_definition.
```

のような宣言に対し、型名を変数に置き換えて次のような CIL 単一化を行う。

```
Man = {name/_, birthday/Date, father/Man}.
Date = {year/_, month/_, day/_}.
Man = {address/_}.
```

その結果、変数 Man,Date に対していくつかの CIL 部分項操作述語を適用することによって、man 型、date 型の構造がそれぞれ man(_,_,_,_) 、date(_,_,_) であることがわかる。従ってプログラム中の X@@man は man(N,B,F,A) という構造に変換可能である。また、{name/john,birthday/{year/1961,month/august,date/15}@@date}@@man も同様にして、man(john,date(1961,august,15),_,_) に変換できる。

継承情報の収集と変換結果

継承行列図2は、継承宣言を表現する継承木図1毎に作る。同じ継承木の中に含まれる型は同じグループとみなされ、同じ継承行列（スキーマ）が与えられる。同じグループ内の型は、その他の型との単一化可能性を表す、継承情報の引数の値で区別される。

```
student < person.
employee < person.
workstudy < student.
workstudy < employee.
peter < student.
mary < workstudy.
paul < employee.
```

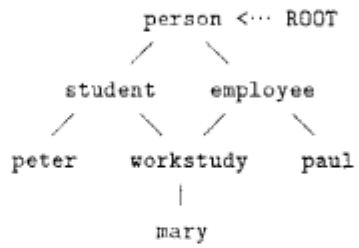

図1 継承宣言と継承木の例

```
          ROOT  per, stu, emp, wor, pet, mar, pau
 person {per,  per,   _,   _,   _,   _,   _,   _}
 student {per,   _, stu,   _,   _,   _,   _,   _}
employee {per,   _,   _, emp,   _,   _,   _,   _}
workstudy {per,  _,   _,   _, wor,   _,   _,   _}
   peter {per,   _,   _,   *,   *, pet,   _,   _}
    mary {per,   _,   _,   _,   _,   *, mar,    }
    paul {per,   _,   *,   _,   *,   *,   *, pau}
```

図2. 継承行列

継承行列の一つの行は、一つの型の継承情報、すなわち他の型との単一化可能性を示す情報を表している。例えば、employee 型の項 person(_,_,emp,_,_,_,_) は、paul 型の項 person(_,*,_,*,*,*,pau) と単一化できるが、peter 型の項 person(_,_,*,*,pet,_,_) とは単一化できない。これは、単一化可能性チェックを満たしている。また paul と単一化された employee 型の項は、person(_,*,emp,*,*,*,pau) の paul 型となり、以降、peter 型の項とは単一化できなくなる。これは、下位型との単一化も含めたデータ型の計算を満たしている。また employee 型と student 型の項の単一化の結果 person(_,stu,emp,_,_,_,_) は、workstudy 型で paul 型や peter 型の項とは単一化できなくなる。これは、多重継承機能を満たしている。

1次元の情報であるこの継承情報も4.7.2で述べたラベル情報と共にエルプラン項に埋め込むことができる。これにより継承機能付きの部分項もエルプラン項に変換できることがわかる。従って、部分項は次のようなエルプラン項に変換される。

```
{ROOT, ..........., ...........}
       継承情報,  ラベル情報
```

型宣言と CIL プログラム

1. 型宣言仕様

<型宣言> ::= <型宣言開始>, {<継承定義> | <型定義>} ··· [<デフォルト型宣言>], <型宣言終了>
<型宣言開始> ::= “begin_definition.”
<型宣言終了> ::= “end_definition.”

<デフォルト型宣言> ::= “default_type(”, <型名>, “).”

<継承定義> ::= <上位型>, “>”, <下位型> | <下位型>, “<”, <上位型>
<上位型> ::= <型名> | “[”,<型リスト>,“]”
<下位型> ::= <型名> | “[”,<型リスト>,“]”
<型リスト> ::= <型名>, {<型名>} ···

<型定義> ::= <型名>, "<=", [<継承付き型用部分項>, <型用部分項>], "."

<継承付き型用部分項> ::= <型用部分項>, "@@", <上位型名>

<上位型名> ::= <型名>
<型名> ::= <アトム>

<型用部分項> ::= "{", <要素対リスト>, "}"
<要素対リスト> ::= <要素対>, {<要素対>} ···
<要素対> ::= <ラベル> , "/", <要素>

<ラベル> ::= <変数なし通常項>
<要素> ::= <型名> | <値> | <値変数> | <継承付き型用部分項>

<値> ::= <通常項>
<値変数> ::= <変数>

注意: 以下の点に注意すること。

- <型名>について
  現在は<アトム>のみを許す。変数付き型は許さない。
- <要素>について
  - 型宣言内ではデフォルト型が宣言できないため<型用部分項>は許されない。
  - "型変数" は許さない。
- <通常項>について
  - <値>としての<通常項>は、その引数の中に<型名>および<型用部分項>を含まない。
    {name/john,hope/work_at(man)} は、 man が<型名>の場合許されない。
  - <ラベル>の<変数なし通常項>は、その引数の中に<変数>, <型名>および<型用部分項>を含まない。
    {name/john, address(X)/osaka} は、ラベルとして<変数>を含むので許されない。

2. 型を用いたCILプログラム

(a) 部分項記法

- 全ての部分項には型名を明記する。
  例　{name/ben}@@man
- 部分項の値として、部分項を記述する場合も型名を明記する。
  例　{father/{name/johnson}@@man}@@man
  但し、暗黙型(default_type)として定義している型は省略可能である。

(b) !記法

- !記法の左側が（ネストしている場合は最も左側が）変数の場合には、型名を明記する。
  例　John@@man!father!birthday
- 左側が（ネストしている場合は最も左側が）部分項の場合には、型名を明記する。（2a部分項記法に従う）
- 暗黙型として定義している型は省略できない。

(c) 変数

- 変数に対する型の明記の場合は節内で有効なので、同一変数に対しては一度でよい。
  例　p(John@@man) :- q(John!father).
  p(John#{name/john}@@man) :- p(John!name).

3. 型宣言の例

```
begin_definition.
   man      <= {name/_, birthday/date}.     % 他の型名の参照
   date     <= {year/_, month/_, day/_}.
   man      <= {address/_, father/man}.     % 帰納型
                                            % 同じ型(man)の定義は併合
   student          <= {course/_}@@man.     % 継承付き型定義
   computerStudent <= {course/computer}@@student.
   date1988 <= {year/1988}@@date.           % 値付き型
   man1988 <= {name/_string, birthday/{year/1988}@@date}@@man.
                                            % 値付き継承付き型 → date の下位型が内部生成
   man1988 <= {name/_string, birthday/date1988}@@man.
   currentStudent  <= {wants/Want, buy/Want}@@student. % 値が変数
   likes <= {  male/{birthday/{month/M}@@date},
             female/{birthday/{month/M}@@date} }.      % 値が変数
   currentComputerStudent <= {computer/Maker,future/work_at(Maker)}@@computerStudent.
   currentComputerStudent <= {computer/Maker,future/{work/Maker}@@dream}@@computerStudent.
                                            % 値が構造体
   dream <= {work/_, marriage/_}.
   default_type(man).                       % 暗黙型(省略型)
end_definition.
```

4.7.3 使用方法

ツールは、CIL シンタックスシュガーを使っているので、宣言ファイルをインクルードするだけで良い。

1. プログラムの記述

   (a) インクルード宣言文を記述する。
   :−include_declaration(toolbox/partial).

   (b) 1型宣言仕様に従って型宣言を記述する。

   (c) 2に従って型を用いた CIL プログラムを記述する。

2. プログラムのコンパイル
   コンパイル、コンサルト等のオペレーションに関しては通常の CIL プログラムと同じ。但し、コンパイル時に public オプションを off(デフォルト時は off) になっていなければならない。

3. ファイル

   宣言ファイル: partial.dcl

   例題ファイル: partial_test1.cil, partial_test2.cil

4.7.4 使用例

以下に例題ファイルの実行例を示す。例題ファイルは二つある。それぞれのファイルをコンパイルして、コンサルトしてから実行する。

1. discourse をツールを使って書き換えた例

```
CIL>compile_program.
  compiling(partial_test1.cil.1) ...
    Including declaration(partial.dcl.1) ... Included
  Compiled 00:00:42
CIL>consult_program.
```

```
  Consult_on(partial_test1.cmp.1) ... Completed 00:00:00
CIL>discourse(I,T).
  I => [soa(love,(jack,betty,loc(1)),yes),soa(love,(betty,jack,loc(2)),yes)]
  T => [A,2]
  yes
```

2. LOGIN の例

```
CIL>compile_program.
  compiling(partial_test2.cil.1) ...
    Including declaration(partial.dcl.1) ... Included
  Compiled 00:00:56
CIL>consult_program.
  Consult_on(partial_test2.cmp.1) ... Completed 00:00:00
CIL>query(X).
  X => 'Bekila'
  yes
```

4.8 ML

4.8.1 概要

MLはLFCと呼ばれる証明検証系を記述するために開発された関数型言語で、証明検証系の開発経験から強力な型推論機能を持つ言語として設計された。MLでは、パラメータ付き多相型(parametric polymorphic type)と抽象データ型(abstract data type)を導入しており、このテーマに関しては、昭和62年度に調査・実験を行いその成果を報告した[11]。MLにおける型推論機能はツールとしての流用性が高いと思われるので、今年度は、昨年度試作したPrologプログラムをCILツールボックスに移植・登録した。

4.8.2 実現方法

ML[12],[11]をCIL上に実装するにあたり、型推論規則としてアルゴリズムWの型推論規則[13]を使用した。その際、関数の再帰的定義を実行するletrec宣言の実現のため、予めYコンビネータの型スキームを

$$\forall\alpha.(\alpha \to \alpha) \to \alpha$$

として与えると共に、アルゴリズムWの型推論規則に、以下に示すLETREC規則を追加した。

LETREC:

$$\frac{\Gamma \vdash (let\ x = e\ in\ (Y.(\lambda x.e'))) : \tau}{\Gamma \vdash (letrec\ x = e\ in\ e') : \tau}$$

つまり、letrec宣言の解釈・実行部では、与えられた型定義にYコンビネータを追加してからLET規則を適用している。また、抽象データ型(abstract data type)の実現のためlettype,abstype宣言の解釈・実行部を追加した。

型宣言の集合Γとしては、CILの述語定義領域(Prologのクローズ・データベースに相当)を利用している。従って、型宣言の追加・変更の際には組込述語assert/retractを使用している。

4.8.3 使用方法

このツールは、ML処理系を記述したプログラムとMLを立ち上げた時に宣言するオペレータの宣言ファイルからなっている。ツールボックスのコンサルト機能を使ってプログラムをコンサルトしてから、CILのトップレベルからpipeコマンドを使ってオペレータを宣言する。トップレベルからmlと入力して実行すると、MLが起動する。

1. ファイル

   ソースファイル: ml.cil

   宣言ファイル: ml.dcl (トップレベルオペレータ宣言用) [2]

2. 述語

   ml/0: ML起動述語
   MLを起動する。

4.8.4 使用例

以下にMLの立ち上げと例題の実行の仕方を示す。まず、MLを立ち上げる。ツールボックスからツールをコンサルトする。

```
CIL>consult_toolbox.
  Consult_on(ml.cmp.1) ... Completed 00:00:01
```

コンサルトが完了したら、MLで使用するオペレータを定義する。

[2]MLで使用するオペレータの内の幾つかはCILとESPのそれと衝突しており、宣言の際にタイプ優先度を変更している。ML終了後、オペレータ宣言をCIL立ち上げ時の状態に戻すには、トップレベルからclear_operatorを実行しなければならない。

```
CIL>pipe(">sys>cil_toolbox>ml.dcl").
CIL>:-op(900,fx,let).
          :    オペレータの宣言が続く。
```

MLを立ち上げる。

```
CIL>ml.
ml>
```

プロンプトが変わりMLの実行環境になる。MLの実行例としてパラメータ付き多相型の例としてmap関数の定義と実行結果を示す。

1. 関数fの定義。組込み関数*(積)が整数の対を整数に写すものとして定義されていることから、システムはfの型が整数を整数に写す関数であることを推論する。

```
ml> let f= \(n,n*n).
f= \(n,n*n) : (int->int)
```

2. 関数mapの定義。*,**は型変数である。

```
ml> letrec map= \(f,\(l,if(null^l,[],[f^(hd^l)|map^f^(tl^l)]))).
map=map@ \(f,\(l,if(null^l,[],[f^(hd^l)|map^f^(tl^l)])))
                                        : ((**->*)->(list(**)->list(*)))
```

3. 関数mapに関数fを適用する。結果の型は[整数のリスト]を[整数のリスト]に写す関数である。

```
ml> map^f.
\(l,if(null^l,[],[\(n,n*n)^(hd^l)|map@ \(f,\(l,if(null^l,[],[f^(hd^l)|
map^f^(tl^l)])))^ \(n,n*n)^(tl^l)])) : (list(int)->list(int))
```

4. 関数mapに関数fを適用して得られた関数に、リスト[1,2,3]を適用する。

```
ml> map^f^[1,2,3].
[1,4,9] : list(int)
```

MLを終了する。

```
ml>exit.
yes
CIL>
```

MLプログラムの実行が終わり、CILトップレベルに戻る。

参考文献

[1] M. Dincbas etc,Extending Equation Soliving and Constraint Handling in Logic Programming,ECRC Internal Report IR-LP-2202,Feb. 1987

[2] 近藤省造, 今村誠, 有限領域を対象とした単一化の拡張,ICOT TM-468,1987

[3] 坂井, 相場,CAL: 制約論理プログラミングの理論と実例, 情報処理学会研究会,SS87-28,1987

[4] A. J. Milner,A theory of type polymorphism in programing, JCSS,17,pp.348-375,1978

[5] 向井, 型推論と談話理解モデル, 情報処理学会第 35 回全国大会,1987

[6] 横田, 型推論と ML,bit,Mar 1988

[7] A. V. Aho,etc., The design and analysis of computer algorithms,Addison-Wesley publishing company, 1974

[8] H. Aït-Kaci & R. Nasr, LOGIN: A Logic Programming Language with Built-in Inheritance,The journal of Logic Programming, 1986:3:185-215

[9] M. Huber, Extended Prolog for Order-Sorted Resolution, Proc. of SLP 87, 1987

[10] G. Montini, Efficiency Considerations on Built-in Taxnomic Reasoning in Prolog, Proc. of IJCAI 87, 1987

[11] 昭和 62 年度 発電設備診断システムの開発成果報告書 要素技術確認実験編,ICOT,3 月 1988

[12] M. J. Gordon, A. J. Milner, C. P. Wadsworth, Edinburgh LCF, Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg NewYork, 1979

[13] 昭和 63 年度 発電設備診断システムの開発成果報告書 要素技術確認実験編,ICOT,3 月 1989