ICOT Technical Memorandum: TM-0107

TM-0107

内包論理のプルーフチェッカー
(A Proof Checker for Intensional Logic)

沢 村 一
(富士通)

March, 1985

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

1．はじめに

内包論理（以下ＩＬと記す）の定理の証明をチェックするシステムをPrologで開発した。

内包論理とはもともと、Montagueが自然言語の様々な内包的意味を捉えるのに導入した論理であり〔Montague 74〕、型の論理に基づいた高階の様相論理として特徴づけられる。

本稿では、内包論理の体系としてGallin〔Gallin 75〕によって形式化された体系を参考にして、その証明チェッカーを構成する。ただし、我々は内包論理のメタ定理の証明にも関心があるので、オブジェクト定理、メタ定理の双方の証明が可能なように内包論理のシンタックスを拡大する。このとき、インプリメンテーション言語であるPrologの変数は内包論理のオブジェクトおよびメタ項を指示するのに用いられることになり、メタメタ変数の役割を果たす。

第２章では、内包論理の計算機向きの定義を与え、第３章では、ＩＬプルーフチェッカーの構成とＩＬの形式的な証明のチェックの例を示す。また、Montague文法における式のリダクションの例も示し、ＩＬのプルーフチェッカーがMontague風言語研究の一つのツールとして利用できることを示す。

2．内包論理

2．1．ＩＬの記号

ここでは、ＩＬをSmullyan風の記述形式〔Smullyan 61〕で記述する。

簡単さのために、ここで許される原子記号及メタ記号は含めてすべてPrologの「アトム」として認められるものに制限する。

ILにおける記号にはすべて型がつけられている。　しかしながら、ユーザが計算機に項を入力するときは、それが複合項であるときその型を省略してよい。代わりに、それらの省略された型は型推論によって求める。

各種の原子記号の定義に先立って、必要なオペレータの定義をPrologのオペレータ

宣言を用いて与える。

```
:  op(-,xfx,':').
:- op(-,fx,'*').
```

〔1〕メタ記号

メタ記号は論理について語ったり、定義したりするのに必要である。そして、これらを対象記号と区別する。以下では、型の上を動くメタ変数(<meta-type>) 、型が <type> の変数記号の上を動くメタ変数(<meta-var>)、型が <type> の定数記号の上を動くメタ変数(<meta const>)、型が<type>の項の上を動くメタ変数(<meta term>)を定義する。

〔形式的定義1〕

(1) <meta-type> ::= t<digit>

<digit> ::= 0|1 |2 |3 | ・・・ |8 |9

(2) <meta-var> ::= <simple-var>:<type> |<simple-var><nat>:<type>

<simple-var> ::= u |v |w |x |y |z |f |g |h

<nat> ::= <自然数>

(3) <meta-const> ::= c<nat>:<type>

(4) <meta-term> ::= a:<type> |b:<type> |c:<type>

〔形式的定義2〕meta-type(<メタ型>)、meta-var(<メタ変数>)、meta-const(<メタ定数>)、meta-term(<メタ項>)

(1)
```
digit(T)-> meta-type(tT)
digit(0)
 :
digit(9)
```

(2)
```
simple-var(V) -> type(T) -> meta-var(V:T)
simple-var(V) -> nat(N) -> type(T) -> meta-var(VN:T)
simple-var(u)
 :
simple-var(w)
```

nat(0)

:

(3) nat(N) -> type(T) -> meta-const(cN:T)

(4) type(T) -> meta-term(a:T)

type(T) -> meta-term(b:T)

type(T) -> meta-term(c:T)

これらの定義はILの定理図式、メタ定理を導くのに必要になる。

〔2〕型

〔形式的定義〕 type(<型>)

(1) type(e)

(2) type(t)

(3) type(X) -> type(Y) -> type((X,Y))

(4) type(X) -> type((s,X))

(5) meta-type(T) -> type(T)

〔3〕対象記号

対象変数、対象定数を次のように定める。

〔形式的定義〕object-var(<オブジェクト変数>)、object-const(<オブジェクト定数>)

(1) meta-var(V:T) -> object-var(*V:T)

(2) atom(C) -> type(T) -> not(object-var(*C:T)) -> object-const(*C:T)

〔4〕変数、定数

変数、定数記号はオブジェクト、メタの双方からなる。

〔形式定義〕 variable(<変数>)、const(<定数>)

(1) object-var(*V:T) -> variable(*V:T)

meta-var(V:T) -> variable(V:T)

(2) object-const(*C:T) -> const(*C:T)

meta-const(C:T) -> const(C:T)

〔5〕論理記号

論理記号とその表現、及びその内容的意味。

(1) ＝（等号）、

(2) λ（ lambda 、関数抽象化オペレータ）、

(3) ^（int 、内包化オペレータ）、

(4) ˇ（ext 、外延化オペレータ）、

(5) 適用/ 並置（# 、関数適用オペレータ）、

(6) ¬（not 、否定）、

(7) ∧（and 、連言）、

(8) ∨（or、　選言）、

(9) ⊃（imply 、含意）、

(10) ∀（all 、全称限量子）、

(11) ∃（some、存在限量子）、

(12) □（nec 、必然オペレータ）、

(13) ◇（pos 、可能オペレータ）

これらの形式的定義には、Prologのオペレータ宣言の方法を流用する。

(1) :- op(-,xfx,=).

(2) :- op(-,fx,lambda).

(3) : op(-,fy,int).

(4) :- op(-,fy,ext).

(5) :- op(-,yfx,#).

(6) :- op(-,fy,not).

(7) :- op(-,xfy,and).

```
(8)    :- op(·,xfy,or).
(9)    :- op(-,xfy,imply).
(10)   :- op(-,fx,all).
(11)   :- op(-,fx,some).
(12)   :- op(-,fy,nec).
(13)   :- op(-,fy,pos).
```

2．2　ＩＬの項

〔１〕項 term(<項>, <型>)

以下の項の定義では、Prolog変数をILのオブジェクト項、メタ項、及び型を表すメタメタ変数として用いている。　term(<項>, <型>)は「 <項> は型が <型> のterm(の集合に属す) である」と読まれる。

```
(1)  variable(V:T) -> term(V:T,T)
(2)  const(C:T) -> term(C:T,T)
(3)  meta-term(Term:T) -> term(Term:T,T)
(4)  term(A,(T,S)) -> term(B,T) -> term(A # B,S)
(5)  term(A,T) -> variable(X:S) -> term(lambda(X:S,A),(S,T))
(6)  term(A,T) -> term(B,T) -> term(A = B,t)
(7)  term(A,T) -> term(int(A),(s,T))
(8)  term(A,(s,T)) -> term(ext(A),T)
(9)  term(*true:t,t)
(10) term(*false:t,t)
(11) term(A,t) -> term(not(A),t)
(12) term(A,t) -> term(B,t) -> term(A and B,t)
(13) term(A,t) -> term(B,t) -> term(A imply B,t)
(14) term(A,t) -> term(B,t) -> term(A or B,t)
(15) term(A,t) -> variable(X:S) -> term(all(X:S,A),t)
(16) term(A,t) -> variable(X:S) -> term(some(X:S,A),t)
```

(17) term(A,t) -> term(nec(A),t)

(18) term(A,t) -> term(pos(A),t)

〔2〕論理式 formula(<論理式>)

型が t の項は論理式と呼ばれる。

(1) term(F,t) -> formula(F)

2．3 定義

ここでは、各種の補助的な定義を行う。

〔1〕論理記号、定数記号の定義

上ですでに必要な論理記号などは定義されているが、ここではそれらの間の関係を定義としてリストする。

```
(1)  def(*true:t,lambda(*x:t,*x:t) = lambda(*x:t,*x:t))
(2)  def(*false:t,lambda(*x:t,*x:t) = lambda(*x:t,*true:t))
(3)  def(not(a:t),*false:t = a:t)
(4)  def(a:t and b:t,lambda(*f:(t,t),*f:(t,t) # a:t = a:t) =
                    lambda(*f:(t,t),*f:(t,t) # *true:t))
(5)  del(a:t imply b:t,a:t and b:t = a:t)
(6)  def(a:t or b:t,not(a:t) imply b:t)
(7)  def(any(x:tl,a:t),lambda(x:tl,a:t) = lambda(a:tl,*true:t))
(8)  def(some(x:tl,a:t),not(any(x:tl,a:t)))
(9)  def(nec(a:t),int(a:t) = int(*true:t))
(10) def(pos(a:t),not(nec(not(a:t))))
```

〔2〕型推論

IL は型付の論理であるが、ユーザからの入力は、変数、メタ変数、定数、メタ定数に

は ' : ' に引き続いて型あるいは型を表すメタ変数が付けられていることを除き、複合項の型は省略されている。したがって、IIの記号操作を整合的に行うには型の推論が必要になる。

ここでは、項とその項に対して型が推論された項の間に成立する述語type-inferenceを推論すると言う方法で項の型推論を考える。

〔型推論の推論形式による定義〕

(1) variable(V:T)

---

type-inference(V:T,V,T)

(2) const(C:T)T

---

type-inference(C:T,C,T)

(3) meta-term(Term:T)

---

type-inference(Term:T,Term,T)

(4) type-inference(A,C,(R,T)) type-inference(B,D,R)

---

type-inference(A # B,C:(R,T) # D:R,T)

(5) type-inference(A,B,T)

---

type inference(lambda(X:S,A),lambda(X:S,B:T),(S,T))

(6) type-inference(A,C,T) type-inference(B,D,T)

---

type-inference(A = B,C:T = D:T,t)

(7) type-inference(A,B,T)

---

type-inference(int(A),int(B:T),(s,T))

(8) type-inference(A,B,T)

---

type-inference(ext(A),ext(B:(s,T)),T)

(9) type-inference(A,B,t)

---

type-inference(not(A),not(B:t),t)

(10) type-inference(A,C,t) type-inference(B,D,t)

---

type-inference(A and B,C:t and D:t,t)

(11) type-inference(A,C,t) type-inference(B,D,t)

---

type-inference(A imply B,C:t imply D:t,t)

(12) type-inference(A,C,t) type-inference(B,D,t)

---

type-inference(A or B,C:t or D:t,t)

(13) type-inference(A,B,t)

---

type-inference(all(X:S,A),all(X:S,B:t),t)

(14) type-inference(A,B,t)

---

type-inference(some(X:S,A),some(X:S,B:t),t)

(15) type-inference(A,B,t)

---

type-inference(nec(A),nec(B:t),t)

(16) type-inference(A,B,t)

---

type-inference(pos(A),pos(B:t),t)

〔3〕様相的に閉じた項(Modally closed term) mc(<項>, <型>)

様相的に閉じた項とは、その解釈が世界に依存しない項をいう。項を型推論した後、それが様相的に閉じた項であるか否かが調べられる。

〔形式的定義〕

(1) variable(V:T) -> mc(V:T,T)

(2) term(A,T) -> mc(int(A:T),(s,T))

(3) mc(A,(T,S)) -> mc(B,T) -> mc(A:(T,S) # B:T,S)

(4) mc(A,T) -> mc(B,T) -> mc(A:T = B:T,t)

(5) mc(A,T) -> mc(lambda(X:S,A:T),(S,T))

(6) mc(*true:t,t)

(7) mc(*false:t)

(8) mc(F,t) -> mc(not(F:t),t)

(9) mc(A,t) > mc(B,t) -> mc(A:t and B:t,t)

(10) mc(A,t) -> mc(B,t) -> mc(A:t imply B:t,t)

(11) mc(A,t) -> mc(B,t) -> mc(A:t or B:t,t)

(12) mc(A,t) -> mc(all(X:T,A:t),t)

(13) mc(A,t) -> mc(some(X:T,A:t),t)

(14) mc(A,t) -> mc(nec(F:t),t)

(15) mc(A,t) -> mc(pos(F:t),t)

〔4〕束縛変数、自由変数

変数 *x:t1あるいはメタ変数 x:t1 の一つの生起が項 a:t2 において束縛されているとは、それが項 a:t2 の中の lambda(x:t1,...) 、lambda(*x:t1,...) 、all(x:t1,...) 、all(x*t1,...) 、some(x:t1,...) あるいは some(*x:t1,...)という部分項に現れることをいう。そうでなければ、自由であるという。変数 *x:t1 あるいはメタ変数 x:t1 のいかなる生起も項 a:t2 において束縛されている(自由である) とき、単に変数 *x:t1 あるいはメタ変数 x:t1 は項 a:t2 において束縛されている(自由である) という。

これらの概念は、ILの項の帰納的な定義にしたがって定義することができる。ここでは、特に変数X:T が項A:S において自由でないという定義が必要となるので、それを項A:S の構造に関する帰納法によって定義する。

(1) const(C:S) -> not-free(X:T,C:S)

(2) variable(V:S) -> not(X:T == V:S) -> not-free(X:T,V:S)

(3) meta-term(Term:S) -> not(X:T == Term:S) -> not-free(X:T,Term:S)

(4) not-free(X:T,A:(R,S)) -> not-free(X:T,B:R) ->
not-free(X:T,(A:(R,S) # B:R):S)

(5) not-free(X:T,lambda(X:T,B:R):S)

(6) not-free(X:T,A:R) -> not-free(X:T,B:R) -> not-free(X:T,(A:R = B:R):t)

(7) not-free(X:T,A:R) -> not-free(X:T,int(A:R):(s,R))

(8) not-free(X:T,A:(s,S)) -> not-free(X:T,ext(A:(s,S)):S)

(9) not-free(X:T,*true:t)

(10) not-free(X:T,*false:t)

(11) not-free(X:T,A:t) -> not-free(X:T,not(A:t):t)

(12) not-free(X:T,A:t) -> not-free(X:T,B:t) ->not-free(X:T,(A:t and B:t):t)

(13) not free(X:T,A:t) -> not-free(X:T,B:t) >not-free(X:T,(A:t imply B:t):t)

(14) not-free(X:T,A:t) -> not-free(X:T,B:t) ->not-free(X:T,(A:t or B:t):t)

(15) not-free(X:T,all(X:T,A:t):t)

(16) not-free(X:T,some(X:T,A:t):t)

(17) not-free(X:T,A:t) -> not-free(X:T,nec(A:t):t)

(18) not-free(X:T,A:t) -> not-free(X:T,pos(A:t):t)

2．4　ＩＬの演繹体系

〔Ｉ〕 ILの公理

ＩＬの公理は次の６つからなる。

(1) axiom(ax1, *g:(t,t) # *true:t and *g:(t,t) # *false:t =
all(*x:t,*g:(t,t) # *x:t))

(2) axiom(ax2, *x:t1 = *y:t1 imply *f:(t1,t) # *x:t1 = *f:(t1,t) # *y:t1)

(3) axiom(ax3, all(*x:t1,*f:(t1,t2) # *x:t1 = *g:(t1,t2) # *x:t1) =
(*f:(t1,t2) = *g:(t1,t2)))

(4) subst(a:t2,x:t1,b:t1,c:t2) -> axiom(ax4, lambda(x:t1,a:t2) # b:t1 = c:t2)

(5) axiom(ax5,nec(ext(*f:(s,t1)) = ext(*g:(s,t1))) = (*f:(s,t1) = *g:(s,t1)))

(6) axiom(ax6,ext(int(a:t1)) = a:t1)

ここで、subst は基本的記号操作機能として、次のように定義されていものとする。

① not-free(X:T,A:S) -> subst(A:S,X:T,B:T,A:S)

② subst(X:T,X:T,B:T,B:T)

③ subst(A:(R,S),X:T,B:T,D:(R,S)) -> subst(C:R,X:T,B:T,E:R) ->
subst((A:(R,S) # C:R):S,X:T,B:T,(D:(R,S) # E:R):S)

④ subst(A:S,X:T,B:T,D:S) -> subst(C:S,X:T,B:T,E:S) ->
subst((A:S = C:S):t,X:T,B:T,(D:S = E:S):t)

⑤ not(X:T == Y:R) -> not-free(Y:R,B:T) -> subst(A:S,X:T,B:T,C:S) ->
subst(lambda(Y:R,A:S):(R,S),X:T,B:T,lambda(Y:R,C:S):(R,S))

⑥ subst(A:(s,S),X:T,B:T,C:(s,S)) ->
subst(ext(A:(s,S)):S,X:T,B:T,ext(C:(s,S)):S)

⑦ mc(B:T,T) -> subst(A:S,X:T,B:T,C:S) ->

subst(int(A:S):(s,S),X:T,B:T,int(C:S):(s,S))

⑧ subst(A:t,X:T,B:T,C:t) -> subst(not(A:t):t,X:T,B:T,not(C:t):t)

⑨ not-free(*f:(t,t),A:t) -> not-free(*f:(t,t),C:t) ->

subst(A:t,X:T,B:T,D:t) -> subst(C:t,X:T,B:T,E:t) ->

subst((A:t and C:t):t,X:T,B:T,(D:t and E:t):t)

⑩ subst(A:t,X:T,B:T,D:t) -> subst(C:t,X:T,B:T,E:t) ->

subst((A:t imply C:t):t,X:T,B:T,(D:t imply E:t):t)

⑪ subst(A:t,X:T,B:T,D:t) -> subst(C:t,X:T,B:T,E:t) ->

subst((A:t or C:t):t,X:T,B:T,(D:t or E:t):t)

⑫ not-free(Y:S,B:T) -> not(X:T == Y:S) -> subst(A:t,X:T,B:T,C:t) ->

subst(all(Y:S,A:t):t,X:T,B:T,all(Y:S,C:t):t)

⑬ not-free(Y:S,B:T) -> not(X:T == Y:S) -> subst(A:t,X:T,B:T,C:t) ->

subst(some(Y:S,A:t):t,X:T,B:T,some(Y:S,C:t):t)

⑭ mc(B:T,T) -> subst(A:t,X:T,B:T,C:t) ->

subst(nec(A:t):t,X:T,B:T,nec(C:t):t)

⑮ mc(B:T,T) -> subst(A:t,X:T,B:T,C:t) ->

subst(pos(A:t):t,X:T,B:T,pos(C:t):t)

〔注意〕公理4はGallinのＩＬでは次のように定義されている。

λXa A(Xa) Ba = A(Ba).

ただし、A(Xa) は項 Aに型がa の変数Xaが自由に現れることを示し、A(Ba) はそれらの自由な変数を項Baですべて置き換えてできた項を表す。このとき、この置き換えは次の条件を満たさなければならない。

（ｉ）A(Xa) の自由な変数Xaに項Baを代入するとき、Baの中に起こる自由な変数が束縛されてはならない。さらに、次のいずれかの条件が満たされなければならない。

（ii）A(Xa) の自由変数Xaは内包オペレータ ^のスコープ内に現れない。

（iii）項Baは様相的に閉じた式である。

上で定義した公理4は、この公理を項Aaの構造に関する帰納法によって定義したもの

である。これは、「変数Xaが現れる項A(Xa) 」というものを、パターンとして定義しにくいことによっている。　ひとつの可能な代替として、「変数Xaが現れる項A(Xa) 」をメタ表現として、

$$(\lambda Xa\ A)Xa$$

と定義する方法がある。すなわち、代入をλ- 式のβ- 変換に肩代わりさせることである。

実際、ＩＬのメタ定理のいくつかはそのような表現を採用して記述した。例えば、

$$\vdash\ \forall Xa\ A(Xa)\ \Rightarrow\ \vdash\ A(Ba)$$

は、

$$\vdash\ \forall Xa\ [\ (\lambda Xa\ A)\ Xa]\ \Rightarrow\ \vdash\ (\lambda Xa\ A)\ Ba.$$

このような記述方法を用いると、式はいくらか長くなるが普通のパターンマッチッグで式の検索ができるようになる。

〔2〕ＩＬの推論規則

ＩＬの推論規則は、

$$\frac{A:T = B:T \quad C:t}{D:t}$$

という形式をもつ、ここで、D:t は C:t に起こるひとつの A:Tを B:T で置き換えて得られる論理式である。

このような推論規則を手続き的に解釈する。すなわち、論理式 C:t の構造に関する帰納的方法によって推論規則を定義する。

```
replace(A:T = B:T,Formula1,Formula2) ->
                        inference-rule(A:T = B:T,Formula1,Formula2)
```

ここで、replace は基本的記号操作機能の一つとして、以下のように定義されている

ものとする。

① replace(A:T = B:T,A:T,B:T)

② replace(A:T = B:T,C:(S,R),E:(S,R)) ->

replace(A:T = B:T,(C:(S,R) # D:S):R,(E:(S:R) # D:S):R)

③ replace(A:T = B:T,D:S,E:S) ->

replace(A:T = B:T,(C:(S,R) # D:S):R,(C:(S:R) # E:S):R)

④ replace(A:T = B:T,C:R,D:R) ->

replace(A:T = B:T,lambda(X:S,C:R):(S,R),lamoda(X:S,D:R):(S,R))

⑤ replace(A:T = B:T,C:S,E:S) ->

replace(A:T = B:T,(C:T = D:S):t,(E:S = D:S):t)

⑥ replace(A:T = B:T,D:S,E:S) ->

replace(A:T = B:T,(C:T = D:S):t,(C:S = E:S):t)

⑦ replace(A:T = B:T,C:S,D:S) ->

replace(A:T = B:T,int(C:S):(s,S),int(D:S):(s,S))

⑧ replace(A:T = B:T,C:S,D:S) ->

replace(A:T = B:T,ext(C:(s,S)):S,,ext(D:(s,S)):S)

⑨ replace(A:T = B:T,C:t,D:t) ->

replace(A:T = B:T,not(C:t):t,not(D:t):t)

⑩ replace(A:T = B:T,C:t,E:t) ->

replace(A:T = B:T,(C:t and D:t):t,(E:t and D:t):t)

⑪ replace(A:T = B:T,D:t,E:t) ->

replace(A:T = B:T,(C:t and D:t):t,(C:t and E:t):t)

⑫ replace(A:T = B:T,C:t,E:t) ->

replace(A:T = B:T,(C:t imply D:t):t,(E:t imply D:t):t)

⑬ replace(A:T = B:T,D:t,E:t) ->

replace(A:T = B:T,(C:t imply D:t):t,(C:t imply E:t):t)

⑭ replace(A:T = B:T,C:t,E:t) ->

replace(A:T = B:T,(C:t or D:t):t,(E:t or D:t):t)

⑮ replace(A:T = B:T,D:t,E:t) ->

replace(A:T = B:T,(C:t or D:t):t,(C:t or E:t):t)

⑯ replace(A:T = B:T,C:t,D:t) ->

replace(A:T = B:T,all(X:S,C:t):t,all(X:S,D:t):t)

⑰ replace(A:T = B:T,C:t,D:t) ->

replace(A:T = B:T,some(X:S,C:t):t,some(X:S,D:t):t)

⑱ replace(A:T = B:T,C:t,D:t) ->

replace(A:T = B:T,nec(X:S,C:t):t,nec(X:S,D:t):t)

⑲ replace(A:T = B:T,C:t,D:t) ->

replace(A:T = B:T,pos(X:S,C:t):t,pos(X:S,D:t):t)

〔3〕派生規則

証明を短く、理解し易くするためには派生規則をできるだけ使うことが望ましい。例えば、

⊢ a:t1 = b:t1 => ⊢ b:t1 = a:t1

といったメタ定理は、派生規則としてIL定理、定理図式の証明に利用される。

〔4〕証明

ＩＬにおける証明とは、次のように定義される。

ＩＬの証明とは論理式の列であって、その各々は公理であるか、さもなくば推論規則により、より前の論理式から得られた論理式である。そして、論理式 A:t がＩＬにおいて証明可能、あるいはＩＬの定理であるとは、それがＩＬの証明の最後の式であるときをいう。

ここでは、ＩＬの定理、定理図式、さらにメタ定理の証明のチェックを考えているので、証明の概念をいくらか修正する。そして、定理の記述法を以下のように定める。

theorem ｜derived-rule <名前>. ⊢ <論理式>.

proof.

1 ⊢ <論理式> by <justification>.

2 ⊢ <論理式> by <justification>.

```
  n ⊢  <論理式>  by <justification>.
end.
```

ここで、<justification> とは、証明行を正当化するための根拠を示すものであり、次のような根拠の表明が考えられる。

① この式は公理である。(axiom)

② この式は推論規則の適用の結果である。(inference-rule)

③ この式は派生規則の適用の結果である。(derived-rule)

④ この式は仮定である。(assumption)

⑤ この式はすでに証明された定理である。(theorem)

⑥ この式は成立するものとする。(proved)

⑦ この式は証明戦略の適用の結果である。(proof-strategy)

⑧ この式は証明手続きの適用の結果である。(proof-procedure)

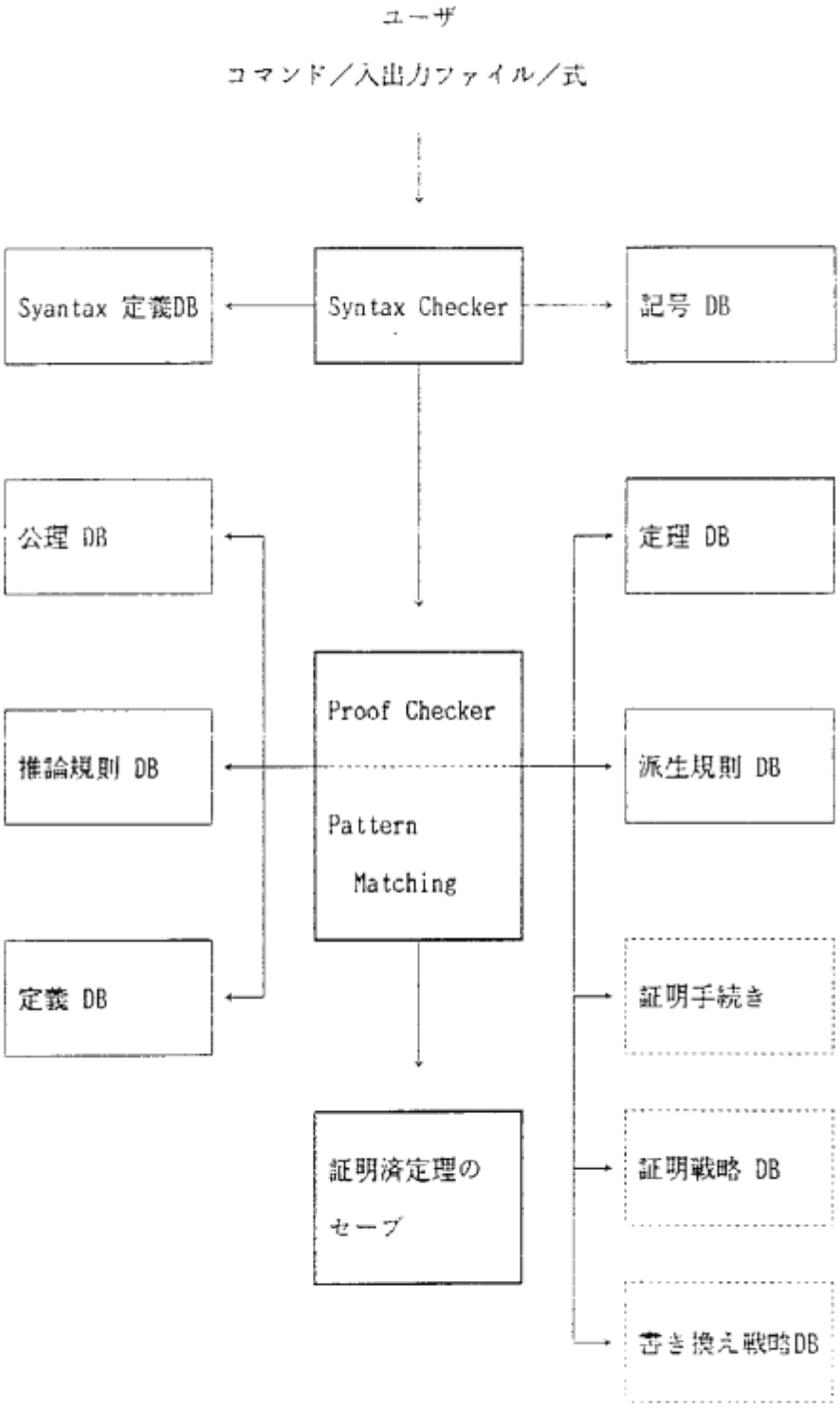
⑨ この式は定義による。(definition)　など

例

```
theorem t1. ⊢ a:t1 = a:t1.
  proof.
    1 ⊢  lambda(x:t1,x:t1) # a:t1 = a:t1  by (axiom,ax4).
    2 ⊢  lambda(x:t1,x:t1) # a:t1 = a:t1  by (axiom,ax4).
    3 ⊢  a:t1 = a:t1  by (inference-rule,1,2).
  end.
```

3．ＩＬプルーフチェッカー

3．1　ＩＬプルーフチェッカーの構成



3．2　ＩＬ証明記述言語

定理、証明は次の形式で入力される。

```
<入力> ::= theorem ｜ derived-rule  <名前>. <定理>.
              <証明>.
<名前> ::=  <Prologアトム>
<定理> ::=  <論理式>
<証明> ::=  proof.
               {< 証明番号> ⊢ <論理式> by <justification>. }
            end.
<証明番号> ::=  <自然数>
<justification> ::= (ir,<証明番号>,< 証明番号>) |
                      ただし、< 証明番号> < 現在の <証明番号>
                    (ax,<公理番号>)  | ax  |
                    (dr,<名前>  {, < 証明番号> } ) |
                      ただし、< 証明番号> < 現在の <証明番号>
                   (th,< 名前>)  | th  |
                    as |
                    def | (def,<証明番号>)
                    proved
                   ( | <証明戦略名> | <書き換え規則名> | <証明手続き名>)
```

3．3　ILプルーフチェッカーの使用法

?- [IL] .

これでILにおける証明のチェックに必要なファイルが consultされる。

?- il-proof-checker (ファイル名).

これでファイルに入っているチェックされるべき証明がILプルーフチェッカーの下

でチェックされる。

また、証明チェックがうまく終えることができた定理などはユーザ指定のファイルにストアすることができる。

3．4　ＩＬ証明チェックの例

〔1〕⊢ a:t => ⊢　∀x:tl a:t (全称一般化定理)

```
derived-rule dr3. ⊢ a:t => ⊢ all(x:tl,a:t).
  proof.
    1.  ⊢ a:t by (as).
    2.  ⊢ (a:t = *true:t) = a:t by (th,t7).
    3.  ⊢ a:t - (a:t = *true:t) by (dr,dr1,2).
    4.  ⊢ a:t = *true:t by (ir,3,1).
    5.  ⊢ lambda(x:tl,a:t) =  lambda(x:tl,a:t) by (th,t1).
    6.  ⊢ lambda(x:tl,a:t) =  lambda(x:tl,*true:t) by (ir,4,5).
    7.  ⊢ all(x:tl,a:t) by (def,6).
  end.
```

〔2〕⊢　∀x:tl a(x:tl):t =　∀y:tl a(y:tl):t　(束縛変数の書き換え定理)

ただし、a(x:tl):t とa(y:tl):t は、次のことを除き同一である: a(y:tl):t は a(x:tl):tの中の自由なx:tlの生起の所にy:tlをもち、a(x:tl):t は a(y:tl):tの中の自由なy:tlの生起の所にx:tlをもっている。

```
theorem t13.  ⊢  all(x:tl,lambda(x:tl,a:t) # x:tl) -
                 all(y:tl,lambda(x:tl,a:t) # y:tl)
  proof.
    1.  ⊢  (lambda(x:tl,lambda(x:tl,a:t) # x:tl) = lambda(x:tl,*true:t)) -
           (lambda(x:tl,lambda(x:tl,a:t) # x:tl) = lambda(x:tl,*true:t))
```

```
        by (th, t1).

2.  ⊢ lambda(x:t1,lambda(x:t1,a:t) # x:t1) =
      lambda(y:t1,lambda(x:t1,a:t) # y:t1)
        by (th,t12).

3.  ⊢ lambda(x:t1,*true:t) # x:t1 = *true:t
        by (ax,ax4).

4.  ⊢ lambda(x:t1,*true:t) # y:t1 = *true:t
        by (ax,ax4).

5.  ⊢ lambda(x:t1,lambda(x:t1,*true:t) # x:t1) =
      lambda(y:t1,lambda(x:t1,*true:t) # y:t1)
        by (th,t12).

6.  ⊢ lambda(x:t1,*true:t) =
      lambda(y:t1,lambda(x:t1,*true:t) # y:t1)
        by (ir,3,5).

7.  ⊢ lambda(x:t1,*true:t) = lambda(y:t1,*true:t)
        by (ir,4,6).

8.  ⊢ (lambda(x:t1,lambda(x:t1,a:t) # x:t1) = lambda(x:t1,*true:t)) =
      (lambda(y:t1,lambda(x:t1,a:t) # y:t1) = lambda(x:t1,*true:t))
        by (ir,2,1).

9.  ⊢ (lambda(x:t1,lambda(x:t1,a:t) # x:t1) = lambda(x:t1,*true:t)) =
      (lambda(y:t1,lambda(x:t1,a:t) # y:t1) = lambda(y:t1,*true:t))
        by (ir,7,8).
```

```
10.  ⊢ all(x:t1,lambda(x:t1,a:t) # x:t1) =
       (lambda(x:t1,lambda(x:t1,a:t) # x:t1) = lambda(x:t1,*true:t))
         by (def).

11.  ⊢ all(y:t1,lambda(x:t1,a:t) # y:t1) =
       (lambda(y:t1,lambda(x:t1,a:t) # y:t1) = lambda(y:t1,*true:t))
         by (def).

12.  ⊢ (lambda(x:t1,lambda(x:t1,a:t) # x:t1) = lambda(x:t1,*true:t)) =
       all(x:t1,lambda(x:t1,a:t) # x:t1)
         by (dr,dr1,(10)).

13.  ⊢ (lambda(y:t1,lambda(x:t1,a:t) # y:t1) = lambda(y:t1,*true:t)) =
       all(y:t1,lambda(x:t1,a:t) # y:t1)
         by (dr,dr1,(11)).

14   ⊢ all(x:t1,lambda(x:t1,a:t) # x:t1) =
       (lambda(y:t1,lambda(x:t1,a:t) # y:t1) = lambda(y:t1,*true:t))
         by (ir,12,9).

15   ⊢ all(x:t1,lambda(x:t1,a:t) # x:t1) =
       all(y:t1,lambda(x:t1,a:t) # y:t1)
         by (ir,13,14).
end.
```

〔3〕 MontagueのPTQ よりの例.

Montague文法では、次の英文

John believes that a fish walks.

はILの式

$\lambda P\ \exists x\ [fish(x) \wedge P\{x\}]\ (\hat{}\lambda y\ [believe(y, \hat{}walk(y))]$

へと翻訳され（タイプは省略する）、この式は次のように簡約されることになる。

$\exists x\ [fish(x) \wedge \hat{}\lambda y\ [believe(j, \hat{}walk(y))\ \{x\}]$ （λ変換）

$\exists x\ [fish(x) \wedge \check{}\hat{}\lambda y\ [believe(j, \hat{}walk(y))]\ (x)]$ （{} 記法）

$\exists x\ [fish(x) \wedge \lambda y\ [believe(j, \hat{}walk(y))]\ (x)]$ （$\check{}\hat{}$消去）

$\exists x\ [fish(x) \wedge believe(j, \hat{}walk(x))]$ （λ変換）

この式の簡約の正当性のチェックをILのプルーフチェッカーによって次のようにできる。

```
theorem t0. ⊢ some(*x:e,(*fish:(e,t) # *x:e and *believe:((s,t),(e,t)) #
                int(*walk:(e,t) # *x:e) # *j:e)).

 proof.
  1 ⊢ lambda(*p:(s,(e,t)),some(*x:e,(*fish:(e,t) # *x:e and
       ext(*p:(s,(e,t))) # *x:e))) #
        int(lambda(*y:e,*believe:((s,t),(e,t)) #
         int(*walk:(e,t) # *y:e) # *j:e))
      by (as).

  2 ⊢ lambda(*p:(s,(e,t)),some(*x:e,(*fish:(e,t) # *x:e and
       ext(*p:(s,(e,t))) # *x:e))) #
        int(lambda(*y:e,*believe:((s,t),(e,t)) #
         int(*walk:(e,t) # *y:e) # *j:e)) =
      some(*x:e,(*fish:(e,t) # *x:e and
       ext(int(lambda(*y:e,*believe:((s,e),(e,t)) #
        int(*walk*(e,t) # *y:e) # *j:e))) # *x:e))
      by (ax,ax4).

  3 ⊢ ext(int(lambda(*y:e,*believe:((s,e),(e,t)) #
       int(*walk*(e,t) # *y:e) # *j:e)) -
```

```
      lambda(*y:e,*believe:((s,e),(e,t)) #
       int(*walk*(e,t) # *y:e) # *j:e)
      by (ax ax6).

4 ⊢ lambda(*p:(s,(e,t)),some(*x:e,(*fish:(e,t) # *x:e and
      ext(*p:(s,(e,t))) # *x:e))) #
       int(lambda(*y:e,*believe:((s,t),(e,t)) #
        int(*walk:(e,t) # *y:e) # *j:e)) =
    some(*x:e,(*fish:(e,t) # *x:e and
      lambda(*y:e,*believe:((s,e),(e,t)) #
       int(*walk:(e,t) # *y:e) # *j:e) # *x:e))
    by (ir,3,2).

5 ⊢ lambda(*y:e,*believe:((s,e),(e,t)) #
     int(*walk*(e,t) # *y:e) # *j:e) # *x:e =
    *believe:((s,e),(e,t)) # int(*walk:(e,t) # *x:e) # *j:e
    by (ax,ax4).

6 ⊢ lambda(*p:(s,(e,t)),some(*x:e,(*fish:(e,t) # *x:e and
      ext(*p:(s,(e,t))) # *x:e))) #
       int(lambda(*y:e,*believe:((s,t),(e,t)) #
        int(*walk:(e,t) # *y:e) # *j:e)) =
    some(*x:e,(*fish:(e,t) # *x:e and
      *believe:((s,e),(e,t)) # int(*walk:(e,t) # *x:e) # *j:e))
    by (ir,5,4).

7 ⊢ some(*x:e,(*fish:(e,t) # *x:e and
      *believe:((s,e),(e,t)) # int(*walk:(e,t) # *x:e) # *j:e))
    by (ir,6,1).
end.
```

4．ＩＬプルーフチェッカーの今後の展開

現在、ＩＬのいろいろな定理の証明のチェックを、ＩＬプルーフチェッカーのサンプルコーディングの下で試みている。今後、使い易さ、効率を上げるために以下の機能拡張が必要である。

①　ＩＬの定理、メタ定理の証明のチェックを行うため、及びPrologの変数との混乱を避けるためにＩＬの式にはすべてPrologのアトムを用いた。このため、記号のパターンマッチ（ユニフィケーションではない）のために多くの時間を必要としている。したがって、Prolog上の効率的なパターンマッチ・アルゴリズムが望まれる。

②　ＩＬの証明を、対話的に一歩一歩作っていけるシステムとする必要がある。これには、公理、定理等のデータベースを自由に参照できる、ウインドゥ風システムが必要である。

③　ＩＬ証明チェックにおいて、証明のステップを簡略化するために証明手続きの利用を可能にする。ＩＬに対しては、例えば文献〔Sawamura 82〕で与えられている様相論理の証明手続きTPPML などが有望である。

④　ＩＬはその公理形のスタイルは等式系である。したがって、等式による簡約法を許すならば証明のステップはかなり短くすることができる。等

謝辞　本研究の一部は第五世代コンピュータ・プロジェクトの一環として行われたものである。日頃御指導、御鞭撻いただく北川敏男会長に感謝いたします。

参考文献

[Ketonen 83] J. Ketonen and J. S. Weening : EKL-An interactive proof checker, User's manual, Dept. of Comp. Sci., Stanford University, 1983.

[Weyhrauch 77] R. Weyhrauch : FOL : A proof checker for first-order logic, AIM-235, Stanford University, 1977.

[Litvintchouk 77] S. D. Litvintchouk and V. R. Pratt : A proof-checker for dynamic logic, 5thIJCAI, 1977.

[Aiello 77] L. Aiello, M. Aiello, et al. : PPC (Pisa Proof Checker) : a tool for experiments in theory of proving and mathematical theory of computation, Annales Societatis Mathematicae Polonae, Series IV : Fundamenta Informaticae I, 1977.

[Constable 81] R. L. Constable, et al. : An introduction to the PL/CV2 programming logic, LNCS,135, 1981.

[Gordon 79] M. J. Gordon, et al. : Edinburgh LCF-A mechanized logic of computation, LNCS, 1979.

[Cohen 81] D. N. Cohen : Knowledge based theorem proving and learning, UMI Research Press, 1981.

[Goldberg 74] A. Goldberg : Desigh a computer-tutor for elememtary mathematical logic, IFIP 74, 1974.

[Smith 75] R. L. Smith et al. : Computer-assisted axiomatic mathematics : informal rigor, in O. Lecarme and R. Lewis (eds.) : Computers in

education, IFIP, N-Holland, 1975.

[Pereira 80] F. C. N. Pereira and D. H. D. Warren : Definite clause grammars for language analysis-A survey of the formalism and a comparison with augmented transition networks, Artificial Intelligence, Vol. 13, 1980.

[Smullyan 61] R. M. Smullyan : Theory of formal system, Princeton University Press, 1961.

[Gallin 75] D. Gallin : Intensional and higher-order modal logic with applications to Montague semantics, North-Holland, 1975.

[Montague 74] R. Montague : Formal philosophy : selected papers of Richard Montague, ed. R. Thomason, New Haven, 1974.

[Sawamura 82] H. Sawamura : Axiomatization of computer-oriented modal logic and decision procedure, Bulletin of Informatics and Cybernetics, Vol. 21, No. 3-4, 1985 (to appear).