修 士 論 文

# 非古典論理に対するシーケント計算における効率的な証明探索法

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

丸山 哲

2002 年 6 月

修 士 論 文

# 非古典論理に対するシーケント計算における効率的な証明探索法


指導教官　小野寛晰 教授

審査委員主査　小野寛晰 教授
審査委員　石原 哉 助教授
審査委員　大堀 淳 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

010113 丸山 哲


提出年月: 2002 年 5 月

# 目 次

**第 1 章　序** **1**

1.1 はじめに . . . . . 1

1.1.1 研究の目的 . . . . . 1

1.1.2 研究の概要 . . . . . 1

1.1.3 内容について . . . . . 2

**第 2 章　シーケント計算 LK と LJ** **3**

2.1 命題と論理式 . . . . . 3

2.1.1 命題 . . . . . 3

2.1.2 論理式 . . . . . 3

2.1.3 部分論理式 . . . . . 4

2.1.4 同等性 . . . . . 4

2.1.5 形式体系 . . . . . 4

2.1.6 始式と規則 . . . . . 5

2.1.7 証明図 . . . . . 7

2.1.8 cut 除去定理と決定可能性 . . . . . 8

2.1.9 終式からの証明探索 . . . . . 11

2.1.10 Wang の体系 . . . . . 12

**第 3 章　Dyckhoff の体系** **14**

3.1 Dragalin の体系 . . . . . 14

3.2 Dyckhoff と Hudelmaier の体系 . . . . . 15

**第 4 章　Dyckhoff の体系の実装** **17**

4.1 使いかたと出力例 . . . . . 17

4.2 アルゴリズム . . . . . 26
4.3 ソースコード . . . . . 27
4.4 プログラムについて . . . . . 34

**第 5 章 まとめ** **36**
5.1 実装について . . . . . 36
5.2 LWB の紹介と比較 . . . . . 37
5.2.1 LWB とは . . . . . 37
5.2.2 LWB の特徴 . . . . . 37
5.2.3 LWB が持つ機能 . . . . . 37
5.2.4 実行例 . . . . . 38
5.2.5 LWB の難点 . . . . . 39
5.2.6 参考点 . . . . . 39

# 第1章　序

## 1.1　はじめに

### 1.1.1　研究の目的

非古典論理に対する定理の自動証明に対してはさまざまなアプローチがある。本研究では cut のないシーケント計算を用いた証明探索について比較、検討を行い、さらに直観主義論理論理の体系に対して Standard ML を用いて計算機への実装をおこなった。

### 1.1.2　研究の概要

よく知られているように cut 除去定理がなりたつ体系の多くにおいては、その subformula property を用いることにより決定可能性を導くことができる。しかしながら、決定可能性を示すために用いられる通常の決定アルゴリズムは LK の場合でさえもきわめて効率が悪い。それは、証明の探索を実行している際に新たに得られた式が、これまでの探索ですでに得られているかどうかをチェックする必要があるからである。このチェックはループチェックと呼ばれるが、ループチェックはメモリと時間の両方を必要とするため効率を下げる大きな要因となる。

古典論理に対しては体系 LK を変形して得られる Wang の体系が知られている。Wang 体系では見かけ上 contraction の規則がなくなっているためにループチェックを必要としない。さらに Wang の体系は上式と下式の証明可能性が同値になっているため、全く機械的な証明探索が可能になり試行錯誤を行なう必要がない。そのため、きわめて効率のよい証明探索が可能になる。

それに対し、直観主義論理の体系に対しては近年になるまでループチェックを必要としないような体系が知られていなかった。1990年代の始めになって Dyckhoff と Hudelmaier が独立にこのような体系を与えることに成功した。この体系では部分的に試行錯誤 (いくつかの選択の可能性) は存在するが従来の方法に比べてはるかに効率がよいものになっ

ている。

この Dyckhoff らによる体系を用いた証明探索法の有効性を確認するため、この研究ではこのアルゴリズムを計算機に実装し、いくつかの実験をおこなった。

### 1.1.3 内容について

本論文は 6 章から構成されている、1 章では本論文の目的と論文構成について述べる、2 章ではシーケント計算の基本的な体系である LK と LJ について、3 章では直観主義論理の体系 LJ の証明探索で生ずるくり返しを取り除いた Dyckhoff の体系について紹介を行う。4 章では、本研究で行った Dyckhoff の体系の実装について工夫した点や出力例を与えながら説明する。5 章ではまとめとして定理自動証明システムとして知られている Logics Workbench (LWB) を紹介するとともに本研究でのシステムとの比較をおこなうという構成になっている。

# 第2章　シーケント計算 LK と LJ

## 2.1　命題と論理式

### 2.1.1　命題

命題とは真偽が確定している文のことである。また、この命題のうち、基本的な命題について記号化、形式化したもののことを命題変数とよぶ、この命題変数は $p, q, r, \cdots$ などの記号を用いることによって表現する。

### 2.1.2　論理式

基本的な命題を論理的操作で結びあわせることによって新しく複合命題を作りだすことができる。この論理的操作を形式化したものを論理結合子と呼び、以下の記号を用いる。

- $\wedge$ かつ 論理積 (conjunction)
- $\vee$ または 論理和 (disjunction)
- $\supset$ ならば 含意 (implication)
- $\neg$ でない 否定 (negation)

これらの論理結合子を用いる事により論理式が定義することができる。
論理式は以下のように定義される。

- $A, B, C, \cdots$ のそれぞれの命題変数は論理式である
- $A, B$ がともに論理式ならば、$(A \wedge B)$, $(A \vee B)$, $(A \supset A)$, $(\neg A)$ いずれも論理式となる

### 2.1.3 部分論理式

論理式 $A$ を構成する論理式のことを部分論理式と呼ぶ。この部分論理式の定義は以下のとおりである。

## 定義

論理式 $A$ の部分論理式は以下のように定義される。

1. $A$ 自身は $A$ の部分論理式である
2. $A$ が $(B \vee C)$, $(B \wedge C)$, $(B \supset C)$ のいずれかの形をしているとき、 $B$ および $C$ の部分論理式はすべて $A$ の部分論理式でもある
3. $A$ が $(\neg B)$ の形をしているとき、 $B$ の部分論理式はすべて $A$ の部分論理式でもある

### 2.1.4 同等性

$(A \supset B) \wedge (B \supset A)$ を $A \equiv B$ と表す。この場合、$A$ と $B$ は同等であると呼ぶ。

### 2.1.5 形式体系

古典論理に対する形式体系として G. Gentzen の sequent 計算の体系がある。ここでは古典論理における sequent 計算の体系として LK と直観主義論理 LJ について説明を行う。体系 LK および、 LJ では基本的な表現は式 (sequent) と呼ばれるもので表現される。
式とは以下のものをいう。

1. $A_1, \cdots, A_m \rightarrow B_1, \cdots, B_n$
2. $\rightarrow B_1, \cdots, B_n$
3. $A_1, \cdots, A_m \rightarrow$
4. $\rightarrow$ $(m = n = 0$ の場合)

この式の直観的な意味としては、それぞれ

1. $A_1$ から $A_m$ までを仮定すると、 $B_1 \vee \cdots \vee B_n$ が導かれる
2. 仮定なしで $B_1 \vee \cdots B_n$ が導かれる
3. $A_1$ から $A_n$ までを仮定すると矛盾が導かれる
4. 無条件で矛盾が導かれる

となる。一方、 LJ では、 LK と同様に定義されるが、 LJ における式は、

$$A_1, \cdots, A_m \to B$$

に限られる。ただし、 $m$ は 0 でもよく、右辺の $B$ がなくても構わない。つまり、 LJ の式は LK の式の定義において右辺に現われる論理式の数を高々1 個に制限したものと考えることができる。

### 2.1.6 始式と規則

LK および、 LJ では公理に相当するものとして始式 (initial sequent) と呼ばれるものを使用する。始式は以下のような形をした式である。

$$A \to A$$

もちろん、 $A$ は任意の論理式である。

sequent 計算では始式にはできる限り簡単なものを取り、その代わりに式の持つ構造や論理結合子のもつ役割を明確に分離してそれら一つ一つを推論規則として表現するという特徴がある。式の持つ構造、とくにコンマの役割に関する推論規則を構造に関する推論規則と言い、それぞれの論理結合子の役割を示す推論規則を論理結合子に関する推論規則と言う。一般に、推論規則は以下のような形をしている。

$$\frac{S_1}{S} \qquad \text{または、} \qquad \frac{S_1 \quad S_2}{S}$$

ここで、 $S_1, S_2, S$ は式である。また、ここでの直観的な意味は $S_1$ がなりたつならば、 $S$ もなりたつ事を示している。これと同様に $S_1$ と $S_2$ がともになりたつならば $S$ もな

りたつ事を示している。この推論規則を I とすると、 $S_1$ および $S_2$ のことを I の上式と呼び、$S$ のことを I の下式と呼ぶ。
また、有限個の論理式をコンマで区切って並べた列を表すのに、$\Gamma$, $\Delta$ などのギリシャ語の大文字を使う。また、これらのギリシャ文字は論理式の列を表すのに使い、アルファベットとは区別される。LK における推論規則は、以下にあげるものをいう。

$$\frac{\Gamma \to \Delta}{\Gamma, A \to \Delta}\ (weakening \to) \qquad \frac{\Gamma \to \Delta}{\Gamma \to \Delta, A}\ (\to weakening)$$

$$\frac{A, A, \Gamma \to \Delta}{A, \Gamma \to \Delta}\ (contraction \to) \qquad \frac{\Gamma \to \Delta, A, A}{\Gamma \to \Delta, A}\ (\to contraction)$$

$$\frac{\Gamma, A, B, \Pi \to \Delta}{\Gamma, B, A, \Pi, \to \Delta}\ (exchange \to) \qquad \frac{\Gamma \to \Delta, A, B, \Sigma}{\Gamma \to \Delta, B, A, \Sigma}\ (\to exchange)$$

$$\frac{\Gamma \to \Delta, A \quad A, \Pi \to \Sigma}{\Gamma, \Pi \to \Delta, \Sigma}\ (cut)$$

$$\frac{A, \Gamma \to \Delta}{A \wedge B, \Gamma \to \Delta}\ (\wedge 1 \to) \qquad \frac{B, \Gamma \to \Delta}{A \wedge B, \Gamma \to \Delta}\ (\wedge 2 \to)$$

$$\frac{\Gamma \to \Delta, A \quad \Gamma \to \Delta, B}{\Gamma \to \Delta, A \wedge B}\ (\to \wedge)$$

$$\frac{A, \Gamma \to \Delta \quad B, \Gamma \to \Delta}{A \vee B, \Gamma \to \Delta}\ (\vee \to)$$

$$\frac{\Gamma \to \Delta, A}{\Gamma \to \Delta, A \vee B}\ (\to \vee 1) \qquad \frac{\Gamma \to \Delta, B}{\Gamma \to \Delta, A \vee B}\ (\to \vee 2)$$

$$\frac{\Gamma \to \Delta, A \quad B, \Pi \to \Sigma}{A \supset B, \Gamma, \Pi, \to \Delta, \Sigma}\ (\supset \to) \qquad \frac{A, \Gamma \to \Delta, B}{\Gamma \to \Delta, A \supset B}\ (\to \supset)$$

$$\frac{\Gamma \to \Delta, A}{\neg A, \Gamma \to \Delta}\ (\neg \to) \qquad \frac{A, \Gamma \to \Delta}{\Gamma \to \Delta, \neg A}\ (\to \neg)$$

LJ の推論規則は LK とまったくおなじものをとるが、 LJ は右辺の論理式を空または 1 個に制限した形でなければならないため、 LJ の推論規則はたとえば以下のようになる。また、 LJ の右辺に対する制限から LJ には ($\to$ contraction) および ($\to$ exchange)

$$\frac{A, \Gamma \to C}{A \wedge B, \Gamma \to C}\ (\wedge 1 \to)$$

$$\frac{\Gamma \to A \quad B, \Pi \to C}{A \supset B, \Gamma, \Pi \to C}\ (\supset \to)$$

$$\frac{\Gamma \to A}{\neg A, \Gamma \to}\ (\neg \to) \qquad \frac{A, \Gamma \to}{\Gamma \to \neg A}\ (\to \neg)$$

$$\frac{\Gamma \to A \quad A, \Pi \to B}{\Gamma, \Pi \to B}\ (cut)$$

図 2.1: LJ の推論規則

は存在しない。論理結合子に関する推論規則を適用したとき、下式に新たに導入される論理式のことをその推論規則の主論理式と呼び、その論理式を構成するために用いられた上式の論理式のことを副論理式と呼ぶ。

### 2.1.7 証明図

始式から推論規則を次々と適用していく過程をすべて書いたものを証明図 (proof figure) と呼ぶ。この証明図を書く過程において上式を 2 つ持つものがあるため、一般には証明図は高々2 つに枝分かれする木の形をしている。また、証明図の一番下にある式を証明図の終式 (end sequent) とよぶ。

## 定義

証明図とその証明図の終式を以下のように帰納的に定義する。

1. 始式はそれだけで証明図であり、その証明図の終式はその始式自身である
2. $P_1$, $P_2$ はそれぞれ、 $S_1$, $S_2$ をその終式とする証明図とする

さらに、

$$\frac{S_1}{S} \qquad \text{または、} \qquad \frac{S_1 \quad S_2}{S}$$

が、推論規則の一つであれば、

$$\frac{P_1}{S} \qquad \text{または、} \qquad \frac{P_1 \quad P_2}{S}$$

は証明図であり、その終式は $S$ となる。

式 $S$ を終式とするような証明図が存在するときは、$S$ は LK もしくは LJ で証明可能 (provable) といい、そのような証明図を $S$ に至る証明図と呼ぶ。式 $\rightarrow A$ が証明可能であるときには、論理式 $A$ が証明可能であると言う。なお、 LJ の証明図はあきらかに LK の証明図であるため、 LJ で証明可能なものは LK でも証明可能である。

### 2.1.8 cut 除去定理と決定可能性

LK, LJ において証明可能な式は cut を用いずに証明可能である。この cut 除去定理はゲンツェンによりはじめて証明された。その方法は、式 $\Gamma \rightarrow \Delta$ の LK において cut を含む証明図が与えられたとき、一つ一つの cut に対してその証明図を変形する具体的な方法を示し、さらにこの変形をくりかえすことにより有限のステップですべての cut が取り除かれることを示すことである。すなわち、これと同様に LJ でも cut 除去定理がなりたつ。

## 定理

式 $\Gamma \rightarrow \Delta$ が LK (LJ) で証明可能ならば、 $\Gamma \rightarrow \Delta$ に至る LK (LJ) 証明図で cut を１度も用いないものが存在する。この (cut) 規則を用いずに証明可能であるということは計算機上へ実装する際に非常に便利である。cut 以外の規則では推論規則の上式に現われるすべての論理式は下式に部分論理式として現われている。このことから次の事が導かれる。

## 定理 (subformula property)

$P$ を式 $\Gamma \to \Delta$ に至る cut なしの証明図とする。このとき、 $P$ に現われるどの論理式も、式 $\Gamma \to \Delta$ に現われるどれかの論理式の部分論理式になっている。

## 決定可能性

任意の体系が与えられたとき、この体系で定義された任意の論理式がこの体系で証明可能であるかを有限回の手順で判定する具体的な手続を決定手続 (decision procedure) と言う。また。その手続が存在する体系は決定可能である (decidable) という。この決定可能性を証明する前に既約について定義する。

## 定義

式 $\Gamma \to \Delta$ について、 $\Gamma$ の中におなじ論理式が高々 3 個しか含まれず、 $\Delta$ のなかにもおなじ論理式が高々 3 個しか含まれないときに、 $\Gamma \to \Delta$ は既約であるという。任意の既約でない式 $S$ に対しては $(exchange \to), (\to exchange)$ および、 $(contraction \to), (\to contraction)$ を何度か用いると既約な式 $S'$ が得られる。また、この既約な式 $S'$ に対しては $(weakening \to), (\to weakening)$ および、 $(\to exchange), (exchange \to)$ を用いることによって元の式 $S$ を導くことができるから、任意の既約ではない式 $S$ にたいし、$S$ と同等でかつ既約な式 $S'$ が存在する。したがって、既約な式のみについての決定手続を与えれば十分なことがわかる。

## 補助定理

既約な式を終式にもつ証明図があれば、これとおなじ終式をもつ証明図が存在して、この証明図に現われる式は全て既約であるようにできる。

## 証明

既約な式を終式とする証明図があるとした場合、始式および終式は既約なためそのままにして、この証明図に現われる各推論規則について次のような変形をおこなう。

推論規則の上式および下式に式 $\Gamma$ があるとき、 $\Gamma$ のなかには一つの論理式は高々 1 回しか現われないように、下式の $\Gamma$ の中の論理式を消し、同時に上式の $\Gamma$ の中の論理式も同じ個数だけ消す。これを、$\Delta, \Pi, \Sigma, \Theta$ などについても同様に行う。この操作を続ける際に上式と下式が同じになれば一方の式を消して、場合によっては cut 以外の構造に関する推論規則を補うことによりなりたつ。

## 定理

LK および LJ は決定可能である。

## 証明

LK または LJ の任意の式 $S$ が与えられたとき、まずこの式と同値で既約な式 $S'$ を求める。$S'$ に含まれる各論理式のすべての部分論理式を数え上げた場合に、部分論理式は有限個しか存在しない。この有限個の部分論理式からつくられる既約な式を全て数え上げるとこれも有限個しかない。これら既約な式全体を $G$ とする。

まず、 $G$ のなかで始式をすべて取り出し、それらは証明可能であると言う印をつける。また、$G$ の残りの式について、各推論規則についてその上式がすでに証明可能として印がついたものであって、しかもその下式が $G$ に含まれる式になっているものをさがす。そして、下式になりうるものについては新たに印をつける。

このような操作を有限回つづけると、 $G$ は有限集合だから、有限ステップの後もう新たに印がつくことのない状況になる。印のついた $G$ の式全体の集合を $G^*$ とする。そこで、始めに与えた式 $S'$ に対し $S' \in G^*$ かどうかをチェックすることにより $S$ が証明可能か否かが判定できる。

### 2.1.9 終式からの証明探索

前節のまでのやり方は始式から終式に向けて、証明図を完成させる方法を示した。しかし、この方法は始式から目的の終式を全探索により求めることになるため非常に効率が悪い。このため、目的の終式から始式へ向けて証明図を書くようにするとこの効率は少し改善されるように思われる。

この終式から始式への証明図の作成であるが、確かに始式から書く場合よりも効率はかなり改善はされる。しかし、計算機上へ実装した場合に規則をくり返し適用してしまうという証明図のループという問題がある。このような探索において cut 以外の推論規則で問題になる惟一の規則は contraction である。実際、contraction

$$\frac{A, A, \Gamma \to \Delta}{A, \Gamma \to \Delta}\ (contraction)$$

において、上式の方が下式より複雑なる (それ以外の規則では、上式の方が、下式よりも簡単な形になる)。そのため、このように上式を予想しながら探索をするという方法では、停止性 (つまり、有限ステップでこれ以上の探索を行なう必要がない状況に達すること) を保証することが難しくなる。それを保証するための一つの方法が先に述べた既約な式に制限する方法であった。

しかしながら、既約な式に制限した場合には、さらに探索において**くり返しが**おこらない、つまり、

$$\cfrac{\cfrac{\cfrac{\vdots}{\Gamma \to \Delta}}{\cfrac{\vdots}{\Gamma \to \Delta}}}{\vdots}$$

という状況が生じないことをチェックする必要がある。これをループチェックという。ループチェックを行うためには、探索のそれぞれの枝において、すでに得られた式をスタックに入れ、新たな式が $S$ が得られたときにスタックに入っているどの式とも $S$ がことなることを確認しなければならない。これは探索を非常に効率の悪いものにする。この問題点を解消するために導入されたのが次節で紹介する Wang の体系である。

### 2.1.10 Wang の体系

古典命題論理 LK と同等であるような体系として Wang の体系がある。この Wang の体系は prover の実装に向いているといわれるため、ここで、その特徴と規則を紹介する。まず、この Wang の体系には以下の特徴がある。

1. $\Gamma, \Delta$ は multiset として考える
2. 始式は $P, \Gamma \to \Delta, P$ の形をしている
3. cut および構造規則をもたない

次に、古典命題論理 Wang の体系の古典命題論理についての規則を以下に示す。
Wang の体系は cut をもたないだけでなく、 weakening rule と contraction rule を持た

$$\frac{\Gamma \to \Delta, A \quad \Gamma, B \to \Delta}{\Gamma, A \supset B \to \Delta}\ (\supset\to) \qquad \frac{\Gamma, A \to \Delta, B}{\Gamma \to \Delta, A \supset B}\ (\to\supset)$$

$$\frac{\Gamma, A, B \to \Delta}{\Gamma, A \wedge B \to \Delta}\ (\wedge \to) \qquad \frac{\Gamma \to \Delta, A \quad \Gamma \to \Delta, B}{\Gamma \to \Delta, A \wedge B}\ (\to \wedge)$$

$$\frac{\Gamma, A \to \Delta \quad \Gamma, B \to \Delta}{\Gamma, A \vee B \to \Delta}\ (\vee \to) \qquad \frac{\Gamma \to \Delta, A, B}{\Gamma \to \Delta, A \vee B}\ (\to \vee)$$

$$\frac{\Gamma \to A}{\Gamma, \neg A \to}\ (\neg \to) \qquad \frac{\Gamma, A \to \Delta}{\Gamma \to \Delta, \neg A}\ (\to \neg)$$

図 2.2: Wang の体系

ない。実際、 weakeing rule はすべて始式に**埋めこまれ**ている。また、 contraction rule は $(\supset\to), (\wedge \to)$ と $(\to \vee)$ の中に**埋めこまれ**ている。このことを Wang の体系の $(\vee \to)$ 規則について確かめてみる。

$$\frac{\Gamma, A, B \to \Delta}{\Gamma, A \wedge B \to \Delta}\ (\vee \to)$$

これは、 LK の体系では、

$$\dfrac{\dfrac{\dfrac{\Gamma, A, B \to \Delta}{\Gamma, A \wedge B, B \to \Delta}\ (\vee \to)}{\Gamma, A \wedge B, A \wedge B \to \Delta}\ (\vee \to)}{\Gamma, A \wedge B \to \Delta}\ (cont \to)$$

のようにして、 contraction を用いることにより得られる。

さらに、 Wang の体系においては、上式は下式よりも必ず簡単な (すなわち、論理結合子の総数が少ない) 形になっている。そのため、停止性が容易に示される。しかも、Wang の体系では、(2 つの) 上式が証明可能であることが同値になっている。したがって試行錯誤なしにつぎつぎに上式を機械的に求めて行けばよい。このことを $(\supset\rightarrow)$ についてみてみる。LK の $(\supset\rightarrow)$ の場合、下式の $\Gamma, \Pi$ および $\Delta, \Sigma$ をそれぞれうまく 2 つに

$$\frac{\Gamma \rightarrow \Delta, A \quad \Pi, B \rightarrow \Sigma}{\Gamma, \Pi, A \supset B \rightarrow \Delta, \Sigma}\ (\supset\rightarrow) \qquad \frac{\Gamma \rightarrow \Delta, A \quad \Gamma, B \rightarrow \Delta}{\Gamma, A \supset B \rightarrow \Delta}\ (\supset\rightarrow)$$

図 2.3: LK と Wang の体系における $(\supset\rightarrow)$ の違い (左：LK, 右： Wang)

分けなければならない。が、 Wang の体系ではその必要がない。このように、 Wang の体系を利用すると非常に効率のよい決定手続が得られることがわかる。

この体系においては、どちらの上式においても上式に現われる論理結合子の総数は下式に現われる論理結合子の総数により必ず小さい。したがって、与えられた式、 $\Gamma \rightarrow \Delta$ から出発して、推論規則を逆に適用してつぎつぎに上式を作りだしていくと有限ステップの後、これ以上推論規則を (逆に) 適用できないような式になる。明らかにこれらの式は、 $p_1, \cdots, p_m \rightarrow q_1, \cdots, q_n$ の形 (ただし、 $p_i, q_j$ はすべて命題変数) である。これらの式がすべて始式になっているか、つまり左辺と右辺に共通な命題変数があるか、をチェックする。すべてが始式の場合は最初の $\Gamma \rightarrow \Delta$ は証明可能であり、一つでも始式になっていないものがあれば $\Gamma \rightarrow \Delta$ は証明可能でないことがわかる。このように Wang の体系を用いた決定手続は効率がよいために実装の際に用いられることが多い。また、古典論理の定理自動証明に用いられているタブロー法は、実質的には Wang の体系と同じものである。しかしながら、この体系を導入する際に用いられたアイディアはそのままでは直観主義論理に用いることはできない。そのために新たな工夫が必要になる。それを次の章で述べる。

# 第3章　Dyckhoff の体系

## 3.1　Dragalin の体系

$$\dfrac{\dfrac{\dfrac{\dfrac{p \to p}{p \to q, p}\,(\to weak)}{\to p \supset q, p}\,(\to\supset) \qquad p \to p}{((p \supset q) \supset p \to p, p}\,(\supset\to)}{\to ((p \supset q) \supset p) \supset p}\,(\to\supset)}{((p \supset q) \supset p \to p}\,(\to cont)$$

$$\dfrac{\dfrac{\dfrac{p \to p, q}{\to p, p \supset q}\,(\to\supset) \qquad p \to p}{(p \supset q) \supset p \to p}\,(\supset\to)}{\to ((p \supset q) \supset p) \supset p}\,(\to\supset)$$

図 3.1: LK と Wang の体系 (左: LK, 右: Wang )

$$\dfrac{}{A, \Gamma \to A, \Delta}\,(action^*) \qquad \dfrac{}{f, \Gamma \to \Delta}\,(f \to^*)$$

$$\dfrac{A, B, \Gamma \to \Delta}{A \wedge B, \Gamma \to \Delta}\,(\wedge \to^*) \qquad \dfrac{\Gamma \to A, \Delta \quad \Gamma \to B, \Delta}{\Gamma \to A \wedge B, \Delta}\,(\to \wedge^*)$$

$$\dfrac{A, \Gamma \to \Delta \quad B, \Gamma \to \Delta}{A \vee B, \Gamma \to \Delta}\,(\vee \to^*) \qquad \dfrac{\Gamma \to A, B, \Delta}{\Gamma \to A \vee B, \Delta}\,(\to \vee^*)$$

$$\dfrac{A \supset B, \Gamma \to A \quad B, \Gamma \to \Delta}{A \supset B, \Gamma \to \Delta}\,(\supset\to^*) \qquad \dfrac{A, \Gamma \to B}{\Gamma \to A \supset B, \Delta}\,(\to\supset^*)$$

図 3.2: Dragalin の体系

Dragalin が [1] で導入した直観主義論理の体系は古典論理に対する Wang の体系と同様なアイディアに基づく。すなわち、 weakening rule は、$(\wedge \to^*), (\to \vee^*)$ および、$(\supset\to^*)$ に埋めこむことにする (この体系では $\neg A$ を $A \supset f$ として定義する。ただし、

これは本質的なものではない。)。 Wang の体系との違いは、 $\supset$ の規則である。まず、$(\rightarrow\supset^*)$ において、上式の右辺はただ一つの論理式でなければならない事に注意してほしい。そのため、 $(\rightarrow\supset^*)$ は weakening を含んだ形にせざるを得ない。つぎに $(\supset\rightarrow^*)$ の左上式は Wang の体系とは異なり $A \supset B, \Gamma \rightarrow A$ の形としておくことが必要である。

明らかに $\wedge$ と $\vee$ の規則については Wang の体系と同様に (2 つの) 上式が証明可能であることは同値であるが、 $\supset$ の 2 つの規則においては下式が証明可能であるからといって上式 (ただし $(\supset\rightarrow^*)$ の場合には左上式) が証明可能になるとは限らない。つまり、 $\supset$ に関しては機械的にではなく「うまく」上式を選ぶ必要が生ずる。

さらに $(\supset\rightarrow^*)$ は停止性を示す際に障害を引き起こす。実際、 $\Delta$ が $A$ より「簡単な」論理式 (の列) である場合には、下式 $A \supset B, \Gamma \rightarrow \Delta$ より上式 $A \supset B, \Gamma \rightarrow A$ が簡単にならなくなってしまうからである。

## 3.2 Dyckhoff と Hudelmaier の体系

この問題の解決のために Dyckhoff と Hudelmaier は別々に同じ体系を与えた ([10] [10])。その体系は、 Dragalin の体系においてその $(\supset\rightarrow^*)$ を以下に示すように 4 つの規則に置き換えることにより得られる。実際にこれらの規則は、下式が $A \supset B, \Gamma \rightarrow \Delta$ のとき、 $A$ がどのような形の論理式であるかによって 4 つに分類されるのである。

$$\frac{B, A, \Gamma \rightarrow \Delta}{A \supset B, A, \Gamma \rightarrow \Delta}\ (\supset\rightarrow_1^*)\text{(ただし、}A\text{は命題変数)}$$

$$\frac{C \supset (D \supset B), \Gamma \rightarrow \Delta}{(C \wedge D) \supset B, \Gamma \rightarrow \Delta}\ (\supset\rightarrow_2^*)$$

$$\frac{C \supset B, D \supset B, \Gamma \rightarrow \Delta}{(C \vee D) \supset B, \Gamma \rightarrow \Delta}\ (\supset\rightarrow_3^*)$$

$$\frac{D \supset B, \Gamma \rightarrow C \supset D \quad B, \Gamma \rightarrow \Delta}{(C \supset D) \supset B, \Gamma \rightarrow \Delta}\ (\supset\rightarrow_4^*)$$

図 3.3: Dyckhoff と Hudelmaier の体系

この体系の停止性を示すには $(\supset\rightarrow^*_2), (\supset\rightarrow^*_3), (\supset\rightarrow^*_4)$ のそれぞれにおいて、(左) 上式が下式より簡単になることを示さなければならない。そのためには、まず論理式 $A$ の重さ $w(A)$ を以下のように定める。

- $w(A) = 1$ ... $A$ が命題変数のとき
- $w(A \vee B) = w(A \supset B) = w(A) + w(B) + 1$
- $w(A \wedge B) = w(A) + w(B) + 2$

すると $(\supset\rightarrow^*_2)$ の下式と上式に現われる論理式の重さはそれぞれ、

$$w(C \supset (D \supset B)) = w(C) + w(D) + w(B) + 2$$

$$w((C \wedge D) \supset B) = w(C) + w(D) + w(B) + 3$$

となる。そして、 $(\supset\rightarrow^*_2), (\supset\rightarrow^*_3), (\supset\rightarrow^*_4)$ のいずれかにおいてもおきかえられる下式の論理式の重さはおきかえて得られる上式の論理式 (2 つの場合もありうる) の重さよりも大きい。このことから、停止性が導かれるのである。(厳密には multiset ordering を利用しなければならない。)

本研究ではこれらの体系を基にした prover の実装を行った。この実装については次章で述べる。

# 第4章　Dyckhoff の体系の実装

本研究で作成したプログラムは前章で述べた Dyckhoff の体系に基づくアルゴリズムを用いて、プログラミング言語としては Standard ML を用いて実装をおこなった。

## 4.1　使いかたと出力例

入力は、プログラムをロードした後の Standard ML より行う。この入力をする場合の論理式の入力例を以下に示す。

- 命題変数は 'Prop' という宣言を先頭に入れる（例：Prop "A"）
- 論理式 $A \wedge B$ は Land (Prop "A", Prop "B") と入力する
- 論理式 $A \vee B$ は Lor (Prop "A", Prop "B") と入力する
- 論理式 $A \supset B$ は Limp (Prop "A", Prop "B") と入力する
- 複数の論理式や命題変数を入力する場合には、それぞれの論理式の間にコンマ ',' を入力する（例: $A \wedge B, C \vee D$ の場合、'Land (Prop "A", Prop "B"), Lor (Prop "C", Prop "D")' と入力）

実際の入力の場合には、この他に証明図を書かせる関数など、一部関数を補ってやる必要がある。この関数などを補う方法については次の入力例を参考にするとよい。

## 入力例

$A \wedge B \rightarrow A \vee B$ を証明させたい場合には、以下のように入力する。

```
main (node ([Land (Prop "A", Prop "B")],([],[])),
node ([Lor (Prop "A", Prop "B")],([],[])));
```

なお、上記の例は紙面の関係上適宜改行してある。
また、 $A \wedge B, C \rightarrow A \vee B$ の場合には以下のようになる。

```
main (node ([Land (Prop "A", Prop "B"), Prop "C"],([],[])),
node ([Lor (Prop "A", Prop "B")],([],[])));
```

# 出力例

出力例として、論理式の証明図を出力させた場合における出力例を以下に示す。

**証明図を出力させたい論理式：**

$$A \wedge B \rightarrow A \vee B$$

**上記の論理式を** prover **へ入力する場合の書き方：**

```
main (node ([Land (Prop "A", Prop "B")],([],[])),
node ([Lor (Prop "A", Prop "B")],([],[])));
```

**出力される証明図：**
上記のシーケントを手書きなどで書いた場合の証明図は以下のようになる。

$$\dfrac{\dfrac{B, A \rightarrow B, A}{A \wedge B \rightarrow B, A}}{A \wedge B \rightarrow A \vee B}$$

この証明図は本研究で作成した prover からは以下のような形で出力される。

```
Node
  ((([Land (Prop "A",Prop "B")],[]),([Lor (Prop "A",Prop "B")],[])),
   [Node
      ((([Land (Prop "A",Prop "B")],[]),([],[Prop "B",Prop "A"])),
       [Leaf (([],[Prop "B",Prop "A"]),([],[Prop "B",Prop "A"]))])]) : tree
```

ここで、出力された結果に関する解説をする。

- 行の先頭の ‘Node’ および、 ‘Leaf’ は、1 段のシーケントを表す
- 特に、最終行の ‘Leaf’ は、始式まで至った場合の木の先端を表す

これより、この例における出力結果は、

1. 2行目は、入力されたシーケントの論理式と命題変数に分かれた結果、つまり、入力されたシーケントそのものを表示している
2. 3行目は、規則を適用した結果、つまり2段目となる
3. 4行目も、規則を適用した結果であるが、木の葉となるためにそれが分かるように 'Leaf' と表示してある

また、木が2つに分かれるときには以下のようになる。

**証明図を出力させたい論理式：**

$$A \vee B \to B, A$$

**出力される論理式：**

$$\frac{A \to B, A \quad B \to B, A}{A \vee B \to B, A}$$

これをプログラムが理解できる形に直すと

**入力する論理式：**

```
main (node ([Lor (Prop "A", Prop "B")], ([],[])),
 (node ([Prop "A", Prop "B"], ([],[]))));
```

**出力される証明図：**

```
Node
    ((([Lor (Prop "A",Prop "B")], []), ([], [Prop "B", Prop "A"])),
     [Leaf (([], [Prop "A"]), ([], [Prop "B", Prop "A"])),
      Leaf (([], [Prop "B"]), ([], [Prop "B", Prop "A"]))]) : tree
```

この証明図をみると、'Leaf' が2つあり、木が2つに分かれていることがわかる。

## 出力が少し複雑になる証明図の場合

$p \vee (q \wedge r) \rightarrow (p \vee q) \wedge (p \vee r)$ の場合

**入力する論理式：**

```
main (node ([Land (Prop"p", (Land (Prop "q", Prop "r")))],([],[])),
node ([Land ((Lor (Prop "p", Prop "q")),(Lor (Prop "p", Prop "r")))],([],[])));
```

**出力される証明図：**

```
  Node
    ((([Land (Prop "p",Land (Prop "q",Prop "r"))],[]),
      ([Land (Lor (Prop "p",Prop "q"),Lor (Prop "p",Prop "r"))],[])),
     [Node
        ((([Land (Prop "p",Land (Prop "q",Prop "r"))],[]),
          ([Lor (Prop "p",Prop "q")],[])),
         [Node
            ((([Land (Prop "p",Land (Prop "q",Prop "r"))],[]),
              ([],[Prop "q",Prop "p"])),
             [Leaf
                (([Land (Prop "q",Prop "r")],[Prop "p"]),
                 ([],[Prop "q",Prop "p"]))])]),
      Node
        ((([Land (Prop "p",Land (Prop "q",Prop "r"))],[]),
          ([Lor (Prop "p",Prop "r")],[])),
         [Node
            ((([Land (Prop "p",Land (Prop "q",Prop "r"))],[]),
              ([],[Prop "r",Prop "p"])),
             [Leaf
                (([Land (Prop "q",Prop "r")],[Prop "p"]),
                 ([],[Prop "r",Prop "p"]))])])]) : tree
```

この場合の証明図は以下のようになる。この場合の出力された証明図の見方を説明する。

まず、‘Leaf’ までが、それぞれの 1 本の枝を示す。この ‘Leaf’ が 2ヶ所でている場合には枝が 2 つに分かれているということである。また、‘Node’ の表示は 1 段終る毎に

$$\dfrac{\dfrac{\dfrac{q \wedge r, p \to r, p}{p \wedge (q \wedge r) \to r, p}}{p \wedge (q \wedge r) \to p \vee r} \quad \dfrac{\dfrac{q \wedge r, p \to q, p}{p \wedge (q \wedge r) \to q, p}}{p \wedge (q \wedge r) \to (p \vee q)}}{p \wedge (q \wedge r) \to (p \vee q) \wedge (p \vee r)}$$

図 4.1: 少し複雑な形になる証明図の例

右へずれて行くが、同じ列から 'Node' が表示される。この場合には、この段から枝分かれしていると言うことである。下図の例では破線を引いてわかりやすくしているが、2 段目から 'Node' が出ているので、2 段目から枝がわかれているということがわかる。

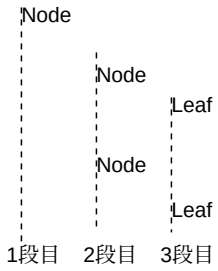


図 4.2: 枝が 2 つに分かれるのを見分ける

このため、サンプルとして挙げた証明図においてもこれと同様に 2 段目からわかれていることがわかる。

## 証明不可能な場合

証明不可能な場合には、プログラムで例外を規定してあるため、その部分が実行され、以下のように表示される。

```
uncaught exception NO_PRINCIPAL
  raised at: /home/*******/*******.sml:90.56-90.68
```

## 入力をまちがえた場合

入力を間違えた場合の処理は規定していないため、その間違いに応じた Standard ML のエラーがそのまま表示される。

## 例 (その1)

- 命題変数に 'Prop' をつけるのを忘れていた場合 (正しい論理式は、前節の論理式とまったく同じものである)

  **間違えて入力してしまった論理式：**

```
main (node ([Land(p, (Land (q, r)))],([],[])),
node ([Land(Lor (p, q)),(Lor (p, r))],([],[])));
```

  **出力されるエラー**

```
stdIn:21.29 Error: unbound variable or constructor: r
stdIn:21.27 Error: unbound variable or constructor: p
stdIn:21.18 Error: unbound variable or constructor: q
stdIn:21.16 Error: unbound variable or constructor: p
stdIn:2.25 Error: unbound variable or constructor: r
stdIn:2.23 Error: unbound variable or constructor: q
stdIn:2.14 Error: unbound variable or constructor: p
```

  この場合には命題変数にそれぞれ 'Prop' をつけてやると正しい証明図が出力されるようになる。

- main 関数自体を付けなかった場合

  **間違えて入力してしまった論理式：**

```
node ([Land (Prop "p", (Land(Prop "q", Prop "r")))],([],[])),
node ([Land ((Lor (Prop "p", Prop "q")),(Lor (Prop "p", Prop "r")))],([],[]));
```

  **出力されるエラー**

```
stdIn:24.54-24.60 Error: syntax error: deleting  RPAREN COMMA ID
```

  この場合には、'main ( 〜間違えて入力した式〜 );' と入力することにより正しい証明図を出力するようになる。

- ‘node’ 関数をつけなかった場合命題変数と論理式を分離させるための ‘node’ 関数をつけなかった場合には次のようになる。

  **間違えて入力してしまった論理式**

```
main (([Land (Prop "p", (Land (Prop "q", Prop "r")))],([],[])),
 ([Land ((Lor (Prop "p", Prop "q")),(Lor (Prop "p", Prop "r")))],([],[])));
```

  **出力されるエラー**

```
stdIn:1.1-24.124 Error: operator and operand don't agree [tycon mismatch]
  operator domain: (formula list * formula list)
                   * (formula list * formula list)
  operand:         (formula list * ('Z list * 'Y list))
                   * (formula list * ('X list * 'W list))
  in expression:
    main
      ((Land (Prop "p",Land (Prop "q",Prop "r")) :: nil,(nil,nil)),
       (Land (Lor (Prop "p",Prop "q"),Lor (Prop "p",Prop "r")) :: nil,
        (nil,nil)))
```

  この場合にはシーケントの両辺に相当する部分へそれぞれ ‘node’ 関数を挿入することにより証明図を出力できるようになる。

## 例 (その 2)

また、カッコの閉じ開きが合わない場合にもエラーが出力される。

- main 関数の閉じカッコがなかった場合

  **間違えて入力してしまった論理式**

```
main (node ([Land (Prop"p", (Land (Prop "q", Prop "r")))],([],[])),
node ([Land (Lor (Prop "p", Prop "q")),(Lor (Prop "p", Prop "r"))],([],[]));
```

**出力されるエラー**

この場合には、 Standard ML は関数の入力が終っていないものと判断してしまうためイコールのみを表示する。

=

この場合には、セミコロン ';' を 3 回入力することにより脱する事が可能である。

- 式中の閉じカッコを付け忘れた場合

**間違えて入力してしまった論理式**

```
main (node ([Land (Prop "p", (Land (Prop "q", Prop "r"))],([],[])),
node ([Land (Lor (Prop "p", Prop "q")), (Lor (Prop "p", Prop "r"))],([],[])));
```

**出力されるエラー**

```
stdIn:24.49-24.52 Error: syntax error: deleting  RPAREN RBRACKET COMMA
```

この場合には、 syntax error と閉じカッコがない旨を表示するため、カッコを正しい位置に挿入することにより証明図を出力するようになる。

これとは別のカッコのつけ忘れによるエラーも紹介する。

**間違えて入力してしまった論理式**

```
main (node ([Land (Prop "p", (Land (Prop "q", Prop "r")))],([],[])),
node ([Land (Lor (Prop "p", Prop "q")), (Lor (Prop "p", Prop "r"))],([],[])));
```

**出力されるエラー**

```
stdIn:38.69-38.121 Error: operator and operand don't agree [tycon mismatch]
  operator domain: formula * formula
  operand:         formula
  in expression:
    Land (Lor (Prop "p",Prop "q"))
```

この場合には、右辺の論理式の一番始まりを ‘Land ((Lor(Prop ”p”, Prop ”q”)))’ と入力しなければならないものをカッコをつけ忘れた例である。この場合にはエラーが出力した一番下の行の ‘Land (Lor (Prop ”p”,Prop ”q”))’ を見てその間違えた部分を正しい形に修正することで証明図が出力できる。

## 4.2　アルゴリズム

本研究におけるプログラムのアルゴリズムは以下のようになる。

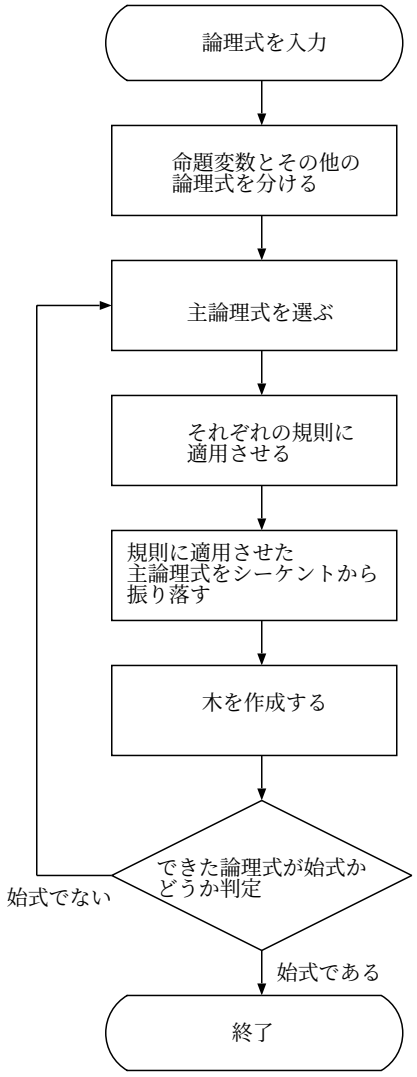


図 4.3: アルゴリズム

説明については次節以降プログラムのソースコードと共に説明を行う。

## 4.3 ソースコード

本研究で作成したプログラムのソースコードは、以下のようになる。

```
(* 長いリスト、文字列を省略させないようにするためのおまじない *)
Compiler.Control.Print.printDepth  := 2000 ;
Compiler.Control.Print.printLength := 2000 ;
Compiler.Control.Print.stringDepth := 2000 ;
Compiler.Control.Print.linewidth   := 2000 ;
(* おまじない終了 *)
```

ここでは、 Standard ML の仕様で長い出力を適宜省略してしまうことがあるが、それを防止するための対策をしている。

```
(* 命題変数、論理式、論理結合子を定義 *)
datatype formula =
Prop of string
| Land of (formula * formula)
| Lor of (formula * formula)
| Limp of (formula * formula)
| False ;
```

ここでは、論理式の型の定義を行っている。まず、この論理式の型は formula 型と名前を決める。まず、命題変数を 'Prop' という名称とし、型は文字列型を示す string 型と定義し、論理結合子 $\wedge$ を 'Land' という名称とし、2 つの命題変数 (論理式の場合も含む) を結びつけている。これと同様に、 $\vee$ では、 'Lor' とし、 $\supset$ では、 'Limp' でそれぞれの論理式の定義を行っている。また、それぞれの論理結合子は、formula 型の組であると定義を行っている。

```
(* シーケントは論理式のリストであると定義 *)
(* 左の (formula list * formula list) は、矢印の左を示し、
 右の (formula list * formula list) は、矢印の右を示す。() 内の
 右の formula list は論理結合子がある論理式を示し、右の formula list
 は命題変数が入るようになっている *)
type sequent = (formula list * formula list) * (formula list * formula list) ;
```

ここでは、シーケントとは、 formula のリストの組の組になっていると定義している。

```
(* 木を定義 *)
datatype tree =
Leaf of sequent
| Node of sequent * tree list;
```

ここでは、木の型の定義を行う。葉は、シーケントであり、それぞれの木のノードはシーケントと、木のリストになると定義する。

```
(* シーケントの両辺におなじものが存在すれば true を返し、なければ
 false を返すようにしている *)

(* まずは、１つの要素とリストを比較して同じものがあれば、 true を返し、
なければfalse を返す関数 *)
(* comp1 : formula * formula list -> bool *)
fun comp1 (x, [])      = false
  | comp1 (x, (y::ys)) =
    if x = y then
true
    else
comp1 (x, ys) ;

(* 次に上の関数を利用して、リスト２つで比較を行う関数 *)
(* comp : formula list * formula list -> bool *)
fun comp ([], _)       = false
  | comp ((x::xs), ys) =
    if comp1 (x, ys) then
true
    else
comp (xs, ys) ;
```

この関数は、シーケントが始式かどうかを判定する関数を作成する際に必要となる関数である。この他には規則に適用させるための主論理式を選び出す際に使用する。

```
(* 命題変数かどうかを判定する関数 *)
(* パターンマッチで実行 *)
(* ip :formula -> bool *)
fun ip (Prop p) = true
    | ip (Land (a, b)) = false
    | ip (Lor (a, b)) = false
    | ip (Limp (a, b)) = false
    | ip (False) = true;

(* 命題変数かどうか判定する関数を使って、命題変数とその他の
 formula list を振り分ける関数 *)
(* node : (formula list * (formula list * formula list)) ->
formula list * formula list *)
fun node ([], (formula_list, prop_list)) = (formula_list, prop_list)
 | node ((x::xs), (formula_list, prop_list)) =
    if ip x then
node (xs, (formula_list, (x::prop_list)))
    else
node (xs, ((x::formula_list), prop_list));
```

これら関数は、上の 'ip' という関数で命題変数かどうかを判定する関数を作っている。この後にある 'node' という関数で命題変数と論理式 (その他の論理式のリスト) を分けている。

```
(* 式の分解を行う関数... 主論理式を見つけ出す為に使う *)
exception NO_PRINCIPAL ;

(* get_principal : sequent -> formula lisrt * formul list *)
fun get_principal ((lhs, lhs_props), (rhs, rhs_props)) =

    let
        (* left_principal : formula list * formula list ->
```

```
formula list * formula list *)
        fun left_principal ([], lhs_props)     = raise NO_PRINCIPAL (*
        左辺に命題変数以外の論理式がなく、命題変数のみの場合には例外を
発生させる *)
          | left_principal (l::lhs, lhs_props) = (*  論理結合子があった場合
        に先頭から切り出す *)
            (case l of
                 (Limp (A, B)) => (* 切り出した論理式のう
        ち、一番外側の論理式に Limp が存在した場合*)
                     (case A of
                          (Prop p) =>(* Limp のつの左側にある論理式が
命題変数の場合*)
                              (if (comp1 (A, lhs_props)) then (* check
つ=>1* - condition ! *)
                                   ([ l ], []) (*つの左側と式の右
        辺に同じ命題変数があった場合、そのまま取りだす *)
                               else
                                   left_principal (lhs, lhs_props))
 (* そうでなければ再度次の要素と比較する *)
                        | _ => ([ l ], [])) (* 命題変数以外の場合には
        そのまま取りだす *)
               | _ => ([ l ], []))(* つ (Limp) 以外の場合にはそのまま取
        りだす *)

        (* right_principal : formula list * formula list ->
 formula list * formula list *)
        fun right_principal ([], rhs_props)     = raise NO_PRINCIPAL
        (* 右辺に命題変数以外がない場合には例外を発生させる *)
          | right_principal (r::rhs, rhs_props) = ([], [ r ]) (*  論理
        結合子があった場合にはそのまま取りだす *)
(* と定義する *)
```

```
    in

        right_principal (rhs, rhs_props) (* まず、右辺から主論理式を選
び始める *)
        handle NO_PRINCIPAL => (* なければ例外を発生する *)
            left_principal (lhs, lhs_props) (* 例外が発生したら、左辺
        の主論理式を始める *)

    end ;
```

この関数は式の分解を行い、規則に適合する主論理式があった場合にはその論理式を取りだす関数である。

```
(* 規則部分 *)
(* ipc_rule : formula list * formula list -> formula list ->
 (formula list * formula list) list *)
exception NO_RULE;
fun ipc_rule (L, R) =
    case (L, R) of
        ([ Land (A, B) ], []) => ("and->", [ ([ A, B ], []) ])
           (* ∧=>* *)
      | ([ Lor  (A, B) ], []) => ("or->",  [ ([ A ], []), ([ B ], []) ])
 (* ∨=>* *)
      | ([ Limp (A, B) ], []) => (* ⊃=>* *)
            (case A of
                 (Prop p)      => ("imp->1", [ ([ B, A ], []) ])
                                  (* ⊃=>1* *)
               | (Land (C, D)) =>
 ("imp->2", [ ([ Limp (C, Limp (D, B)) ], []) ]) (* ⊃=>2* *)
               | (Lor  (C, D)) =>
 ("imp->3", [ ([ Limp (C, B), Limp (D, B) ], []) ]) (* ⊃=>3* *)
               | (Limp (C, D)) =>
 ("imp->4", [ ([ Limp (D, B) ], [ Limp (C, D) ]), ([ B ], []) ]) (* ⊃=>4* *)
```

```
                | _ => raise NO_RULE)
      | ([], [ Land (A, B) ]) =>
 ("->and", [ ([], [ A ]), ([], [ B ]) ]) (* => ∧* *)
      | ([], [ Lor  (A, B) ]) =>
 ("->or",  [ ([], [ A, B ]) ])           (* => ∨* *)
      | ([], [ Limp (A, B) ]) =>
 ("->imp", [ ([ A ], [ B ]) ])           (* => ⊃* *)
      | _ => raise NO_RULE ;
```

この関数は、規則の本体となる関数であり、前の選ばれた主論理式に対して適切な規則を適用させていく。

```
(* get_principal から出たシーケントから規則を適用したものを落して行く関数 *)
(* リストから要素を取り除く補助的関数 *)
fun drop _ [] = []
  | drop x (y::ys) = if (x = y) then ys else (y :: (drop x ys)) ;

(* 落していく関数 *)
(* elim_principal : (formula list * formula list) ->
 (formula list * formula list) -> sequent *)
fun elim_principal (([ l ], []), (lhs, rhs)) = (drop l lhs, rhs)
  | elim_principal (([], [ r ]), (lhs, rhs)) = (lhs, drop r rhs)
  | elim_principal (_, _) = raise NO_PRINCIPAL ;
```

上の drop 関数で規則が適用されたものを論理式のリストから消すための関数となるための準備段階の関数として使い、 elim_principal 関数で、論理式のリストから実際に消していく関数ととして機能する。

```
(* align_side : formula list * formula list -> sequent -> sequent *)
(* 木をつくるための関数 *)
fun align_side (ls, rs) ((lhs_remains, lhs_props), (rhs_remains, rhs_props)) =
    (node (ls, (lhs_remains, lhs_props)), node (rs, (rhs_remains, rhs_props))) ;
```

この関数は、上の elim_principal 関数で論理式のリストから消された後に証明図の木を作成することになるのでその木を作りだす関数として使う。

```
(* シーケントが始式かどうか判定する関数 *)
(* ifaxiom : (formula list * formula list) * (fomula list * formula list) ->
 bool *)
fun ifaxiom ((lhs, lhs_props), (rhs, rhs_props)) =
(comp (lhs, rhs)) orelse
(comp (lhs_props, rhs_props)) orelse
(comp1 (False, lhs_props)) ;
```

この関数は、シーケントが始式かどうかを判定する関数として機能する。ここで、false となると、**始式ではない**ということになる。また、**始式である**場合には、 true を出力する。

```
(* make_node : string * ((formula list * formula list) list) -> sequent -> seque
nt list *)
(* 木を作る関数ではあるが⊃ ->4 において、一部分改編する部分が生じている
   ため、そのための対処 *)
fun make_node ((rule, side_fors),
((lhs_remains, lhs_props), (rhs_remains, rhs_props))) =
    case rule of
        "imp->4" =>
            [align_side (hd side_fors)      ((lhs_remains, lhs_props),
([], [])), align_side (hd (tl side_fors)) ((lhs_remains, lhs_props),
 (rhs_remains, rhs_props))]
      | "->imp"  =>
            [align_side (hd side_fors) ((lhs_remains, lhs_props), ([], []))]
      | _        =>
            map (fn sf => align_side sf ((lhs_remains, lhs_props),
 (rhs_remains, rhs_props))) side_fors ;

(* main となる関数 *)
```

```
fun main (seq as ((lhs, lhs_props), (rhs, rhs_props))) =
  if ifaxiom seq (* 始式かどうかを判定させる *)
      then
  Leaf seq (* 始式であれば、葉になる *)
      else
  let
      val principal = get_principal seq (* 主論理式を選
ぶ *)
      val (rule, new) = ipc_rule principal (* 規則に適用
させる *)
      val (lhs', rhs') = elim_principal (principal,
(lhs, rhs)) (*規則に適用させた論
      理式を落す *)
      val new_tree = (make_node ((rule, new),
     ((lhs', lhs_props),
     (rhs', rhs_props)))) (* 木を作る *)
  in (* と局所的に関数を定義をする *)
      Node (seq, map main new_tree) (* map 関数でリスト
となっているそれぞれのシーケントに適用させる *)
  end;
```

この関数は、すべての関数をまとめあげる役割となるメインの関数である。詳しい動きなどはアルゴリズムの図とまったくおなじ動作をするようになっている。

## 4.4 プログラムについて

このプログラムは、前述した通り Standard ML で作成した。このなかで、論理式および、命題変数などの論理式の根幹に関わる部分については string 型で記述をしてそれを、 formula 型などの prover を作成する際に使いやすいように型を新たに作りだしている。また、それぞれの段になるシーケントを list 型に入れることによって規則に適用する主論理式を調べることを容易にさせている。これと同じように、論理式と命題変数を別の list に入れる事で主論理式を選びやすくなっている。

大まかな部分は、以上であるが、更に細かい部分についてはプログラムのソースコードにコメント文とともに解説文があるのでそれを参照していただくと更に細かくわかるものと思う。

# 第5章　まとめ

本研究では最終的にはシーケントによる効率のよい prover の実装をするという目的として研究をすすめてきた。ここではその実装の際の総括および、今後の課題点等について述べる。

## 5.1　実装について

実装については 4 章でも述べた通り Standard ML で作成した。この Standard ML は定理自動証明システムを作成するに向いている言語のため、関数の定義を論理式の定義と同じようにできたり、型を定義するのが非常にやりやすいものであった。さらに、Standard ML は関数型言語であるため、パターンマッチなどにより論理式や命題変数の特定が容易にできる。そのため定義に基づく実装のみに専念できる点や、別の言語のようにメモリ配分などを考えずに実装を行うことがでる点などにおいて優れている。これらの利点により、私のような他分野から来た人間が論理体系の実装するに専念できたということは非常に嬉しいことでもあった。

また、論理体系としては効率のよい Dyckhoff の体系を用いて直観主義論理の prover の実装を行ったため prover としては有意義なものができたと考えている。

しかし、 Standard ML そのままの実装であるため入力や出力の修得に際してプログラムに合わせた入力方式、出力を読み解くコツと慣れが必要になっている。これらについては使用者に対する入力に関する負担軽減を目的とした parser 等の開発やユーザーにとってより使いやすいものにする工夫が必要であることを痛切に感じた。

## 5.2 LWB の紹介と比較

### 5.2.1 LWB とは

A. Heuerding 氏が作成した prover に Logics WorkBench (以下、 LWB と略す) というものがあり、すでに広く利用されている。この LWB と本研究で作成した prover の比較を通じて、今後の改良すべき方向を考察してみた。

### LWB の入手

LWB の入手は以下の URL からできる。

`http://www.lwb.unibe.ch/`

### 5.2.2 LWB の特徴

この LWB は標準的な UNIX の X-Window system (含む Linux) で動作する。また、実装されている論理は、古典命題論理、直観主義論理、様相論理、多様相論理、時間命題論理、線形命題論理など様々な論理を扱うことができる。また、この LWB では真理値表の出力も可能で、入力は本研究で作成したソフトウェアよりもわかりやすい印象がある。

### 5.2.3 LWB が持つ機能

この LWB の主な機能を挙げると以下のようになる。

1. UNIX の X-Window system (Linux も含む) で動作する
2. 実装している論理が多い (古典命題論理、直観主義論理、様相論理、多様相論理、時間命題論理、線形命題論理など)
3. 真理値表の出力が可能
4. 本研究で作成したプログラムよりも容易に使用できる

### 5.2.4 実行例

実行の流れは簡単に説明すると以下の通りである。

1. 論理体系を選択する
2. 論理式を入力する
3. 真理値表を書く、証明図を出力するなど、実行させたい命令を入力する

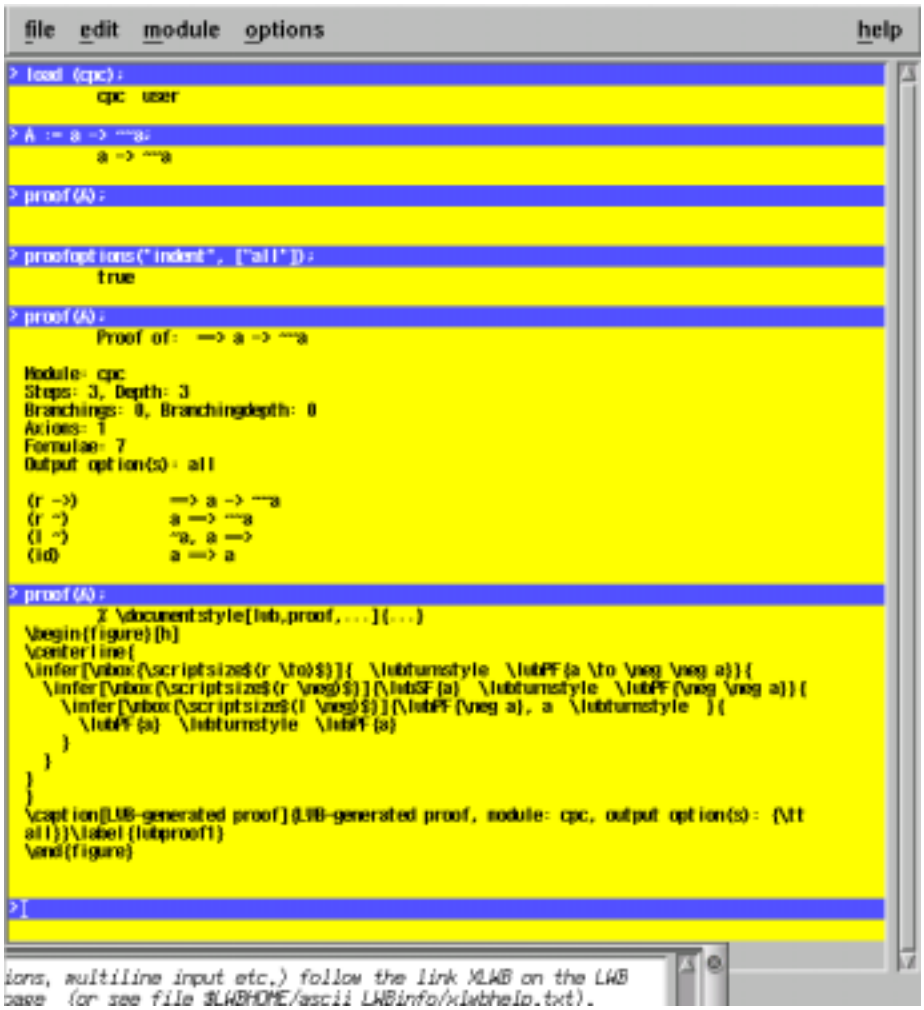


図 5.1: Logics Workbench の画面

図は実際に実行してみた際の実行例である。この図では、以下の事を行っている。

1. ‘load (cpc);’ で命題古典論理を選択
2. ‘A := a ->  a;’ で $a \to \neg\neg a$ を A へ代入
3. ‘proofoptions(”indent”, [”all”]);’ で証明図を書くためのおまじないをする
4. ‘proof(A);’ で A に代入した証明図を出力させる (上は標準的な出力、下は LaTeX での出力を選択した場合)

### 5.2.5 LWB の難点

この LWB だが難点もある。この難点をいくつか挙げてみると、

1. 入力に独自の関数を利用するため修得するに時間とコツが必要である
2. ソースコードが公開されてなく、バイナリのみでの公開であるため、ライブラリ等のバージョンが合わない場合には起動できない事があった (Vine Linux 2.5 がこれに該当した)
3. 標準の場合における画面の色使いがかなり派手なため長時間使用する場合には目が疲れそうな印象がある
4. 論理式によっては証明図を書かせようとすると OS ごと落ちることがある
5. 論理式によっては証明図を書かせようとすると lwb が落ちることがある

以上が私が卒直に感じた難点である。

### 5.2.6 参考点

以上のような特徴を持った LWB であるが、私自身が今後この LWB より参考にして、私自身が利用できそうな点などまとめてみた。

- X-Window system を利用しているだけでなく、 kterm などの端末コンソールからの入力にも対応している
- 他の OS にも対応している (Mac のみ)

- 入力した論理式等をファイルに保存し、必要に応じて再度使用することができる
- 出力方法は入力に対応したままの形だけでなく、 LaTeX などの様々な形式に対応してるらしい
- 様々な論理体系に対応している
- プログラミング言語並みに豊富な命令がある

これらの点は今後開発する上において非常に参考になるものと思われる。

特に、X-Window に対応している点はユーザーインターフェースが親しみのあるもののように感じる。このため、私を含めた LWB の使いかたがまだわからない人に対しては操作方法を直感的に理解ができるものになっている。こまごまとした使用法や出力されて来た内容をどうするかなどを理解する必要なく即使いこなそうとする人に対してはかなり便利なものであると感じた。

さらに、入力した論理式などを保存できると言うことはやりかけの仕事や、入力最中の論理式を保存できるため、とくに長い論理式等を扱う場合にはかなり有用である。

また、出力が LaTeX に対応している点は、 LWB の標準での表示があまり好ましい表示ではないため ($\vee$ を 'v' で入力・表示させるなど)、 LaTeX で出力できるというのは、出力結果をそのまま、論文などへ書く場合にはかなり有用な機能である。

プログラミング言語なみに命令があるので、複雑なものを書くときや、さまざまな論理式を組み合わせた使いかたができそうであり、これは興味がある機能であると感じた。

以上が私が簡単に使ってみた参考点であるが、もっと使うことによってさらに様々な発見などもあるものと考えられる。

# 参考文献


[1] A. G. Dragalin, Mathematical intuitionism - introduction to proof theory, Translations of Mathematical Monographs, vol.67, American Mathematical Society, Providence, Rhode Island, 1998.

[2] R. Dyckhoff, Contraction-free sequent calculi for intuitionistic logic, The Journal of Symbolic Logic 57, 795-807, 1992.

[3] R. Dyckhoff, N. Leslie and S. Read, MALT St Andrews (MacLogic - a proof assistant for first-order logic), Computerised Logic Teaching Bulletin, vol.2, no.1, St Andrews University, 51-69, 1989.

[4] A. Heuerding, Sequent calculi for proof search in some modal logics, PhD thesis, University of Bern, Switzerland, 1998.

[5] J. Hudelmaier, A Prolog and types, Cambrideg University Press, Cambridge, 1989.

[6] J. Hudelmaier, Bounds for cut elimination in intuitionistic propositional logic, Archive for Mathematical Logic, 31, 331-353, 1992.

[7] J. Hudelmaier, An $O(n \log n)$ - Space decision procedure for intuitionistic propositional logic, Jornal of logic and computation. Vol.1, No.1, Oxford University press, 63-75, 1990.

[8] 松本和夫, 数理論理学, 共立講座 現代の数学 1, 共立出版, 1970.

[9] 大堀 淳, プログラミング言語 Standard ML 入門, 共立出版, 2001.

[10] 小野寛晰, 情報科学における論理, 情報数学セミナー, 日本評論社, 1994.

# 謝辞

本研究を行うにあたり、常日頃から熱心にご指導くださいました小野寛晰先生に深く感謝致します。また、貴重なご意見をくださいました石原哉助教授、浜野正浩助手、 Tomasz Kowalski 助手ならびに小野研究室、石原研究室の皆様に感謝致します。また、 Standard ML をまったく知らなかった私に対して懇切丁寧に教えてくださった小野研究室の松本利雅さんに感謝致します。