構成的論理に基づく
プログラムの合成と解析の研究

後藤滋樹

①

構成的論理に基づく
プログラムの合成と解析の研究

1990年11月

後藤 滋樹

もくじ

第 1 章

序論

1.1 本研究の目的

コンピュータ・プログラムを作成することは人間の知的活動の一つである。人間は単にプログラムを書くばかりでなく、出来上がったプログラムの性質を論じることも出来る。このような知的活動をそれぞれプログラムの合成、プログラムの解析と言う。本研究はプログラムの合成や解析を自動化することを長期的な目標としている。ただし、この目標を達成するためには数多くの問題を解決しなければならない。本研究では長期的な目標に至る里程標 (マイルストーン) となるべき幾つかの問題を取り扱い、その解決をはかった。

このようにプログラムの合成や解析を研究対象とする際には、プログラムの仕様やプログラムの性質を記述するための表現形式が必要になる。もちろん新規にプログラミング言語を設計して、プログラムの仕様やその性質までもプログラムの一部として記述できるようになれば、一つの言語だけを対象として議論を進めることが可能になるであろう。しかしそのような言語は未だ研究段階にあり、実現のためには本研究で述べるような表現形式の研究成果を活用しなければならない。

本研究では、専ら論理式を表現形式としてプログラムの合成および解析を論じる。他の表現形式の候補としては自然言語がある。実際、冒頭に述べた人間の知的活動としての合成や解析においては自然言語が重要な役割を果している筈である。しかしながら、良く言われるように自然言語による表現には常に曖昧性が付きまとっている。自然言語の曖昧性の取り扱い自体が情報工学の研究対象になっているのが現状であるから、我々としては論理式の方を選択する。

さて表現形式として論理式を採用するということは、その背後に何らかの論理体系を仮

定することを意味する。なぜなら個々の論理式は論理体系の中で意味を持つからである。似たようなことはプログラムの側にも言える。個々のプログラムはプログラミング言語の中で意味を持つ。この関係を図解すれば図 1.1の様になる。

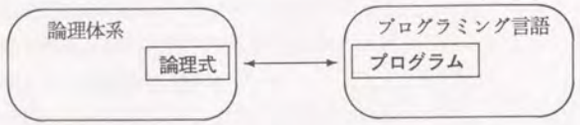

図 1.1: 表現形式としての論理式

ここで注意が必要なのは、論理体系はコンピュータの出現以前から存在するものであり、プログラムの理論以外にも広範に適用されている。すなわち、論理体系はプログラムの世界とは独立に存在しているのである。したがって、論理式を用いてプログラムの合成や解析を行う時には次の点に注意を払う必要がある。

a) 論理体系における各種の概念と、プログラムの世界の対象物・事象とがどのように対応付けられるか。

b) 論理体系の選び方には何通りかの選択があり得る。上の対応付けは論理体系よって異なると思われるが、どの論理体系がプログラムの合成や解析を研究するのに最も適しているか。

上記の a) については、本研究以前にも多くの研究成果がある。その結果をまとめると、先の図 1.1で論理式とプログラムとが対応するように描いた部分をさらに詳細化できる。すなわち論理式に直接に対応するのはプログラム本体でなく、プログラムの仕様である。仕様を表す論理式が証明可能である時、その証明がプログラム本体に対応する。証明の中には代入 (substitution)、場合分け (case analysis)、帰納法 (recursion) 等が現れる。この情報を抽出し、適切なプログラミング言語で表現すればプログラムが得られる。なお正確に言えば、証明の方がプログラムよりも若干情報量が多い。このような証明とプログラムとの対応付けは本研究の基盤を成すものであるから、次節でやや詳しく述べることにする。

本研究の特徴は b) の問題を本格的に扱った点にある。すなわち、従来の研究においては論理体系として古典論理 (classical logic) を用いるのが普通であった。本研究では、古

典論理とプログラムとの対応付けの問題点を指摘し、プログラムの合成や解析のためには直観主義論理 (intuitionistic logic) に代表される構成的論理 (constructive logic) を使うべきであることを主張している。古典論理の問題点を簡単に述べておこう。

問題点:

i) 上で論理式の証明がプログラムに対応すると説明したが、一般に論理式の証明の仕方は一通りではない。各々の証明に一つずつプログラムが対応するから、同一の仕様を満たすプログラムは何通りにも書ける。ここで古典論理に基づいた対応関係を用いると、論理式の証明の中にプログラムとしては解釈できない証明が出てくる可能性がある。言い換えると、古典論理ではすべての証明とプログラムとの対応が付く訳ではない。(図 1.2参照)

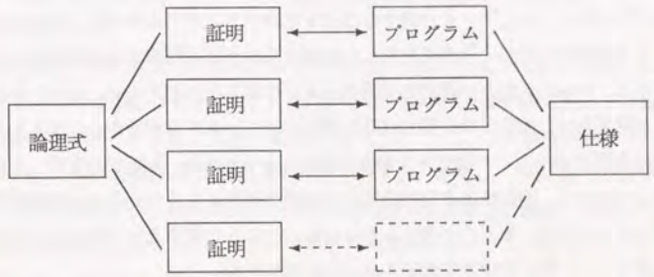

図 1.2: 古典論理の問題点 (i)

ii) 二つの論理式が同値であっても、対応するプログラム同士が同値になるとは限らない。ここに二つの論理式 A と B とが同値であるとは、A⊃B (A ならば B) と B⊃A (B ならば A) とが同時に成り立つことを言う。また二つのプログラム $P_1(x)$ と $P_2(x)$ とが同値であるとは、すべての入力 x に対して出力の値が等しいこと ($P_1(x)=P_2(x)$) を言う。(図 1.3参照)

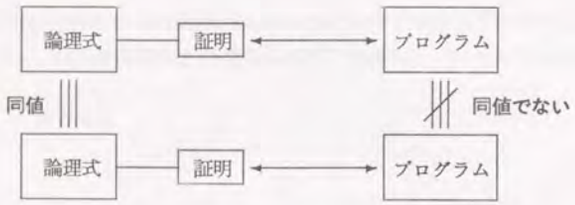

図 1.3: 古典論理の問題点 (ii)

いずれにしても、古典論理の抱える不整合の問題は直観主義論理を用いると解消する。この点は第 2 章で詳しく論じる。

上述の様に、直観主義論理を用いるとプログラムの世界の対象物・事象と論理体系との整合性が良くなる。本研究では、さらに進んで仕様を表す論理式の証明から直観主義論理に基づいて具体的にプログラムを合成する方法を考察する。そのためには、従来から行なわれている古典論理に基づく合成法を新しい論理体系の下で再構成する必要がある。このように書くと、いかにも新しい理論を構築するように響くかも知れない。しかし本研究で活用する基礎理論は、論理学の分野では既に確立されていたことが分かる。すなわち、古い時代に論理学者が主として論理体系の**無矛盾性** (consistency) の証明のために考案した種々の手法の中には、論理体系を帰納的関数の世界で解釈するものがある。帰納的関数はプログラムをモデル化したものと考えられるから、これらの成果をコンピュータの分野で適切に応用すれば、我々の研究を前進させるのに役に立つ。

表 1.1には本論文で用いた理論的な手法をまとめておいた。いずれも論理学の分野では著名な業績として知られている。もちろん論理学における無矛盾性の証明は純理論的な成果である。したがって「無矛盾性が証明できる」という事実そのものが直ちにプログラムの世界で意味を持つわけではない。しかし証明の過程で用いられた論理式の解釈法は、いずれも論理式とプログラムとを対応付ける技法として役に立つ。本論文は、このような立場で論理学の成果をプログラムの合成と解析に応用することを目標とした。その意味で、本研究は言わば応用論理学である。

本論文の第 2 章では Kleene が提案した realizer という手法を応用して論理式とプログラムとの対応付けを考える。この対応付けは直観主義論理に基づいており、前述の古典論理の問題点を解消するものである。

手法の名称	理論的な論文	プログラムのモデル	本論文の該当箇所
realizer	Kleene [27]	帰納的関数	2 章
ゲーデル解釈	Gödel [16]	原始帰納的汎関数	3 章
証明図の正規化	Prawitz [43]	ラムダ計算	4 章、5 章

表 1.1: 本論文で用いた論理的な手法

本論文の第 3 章では表 1.1の 2 行目にあるゲーデル解釈 (Gödel Interpretation) という論理学の手法を応用して実際にプログラムを合成する手法を考案した。この表 1.1の中にある原始帰納的汎関数 (primitive recursive functional) というのは計算の理論で良く使われる原始帰納的関数を若干拡張したものである。ゲーデル解釈の理論は昔から論理学の分野では知られているが、本研究では実際にプログラムの合成システムを Lisp を用いてインプリメントし、幾つかのプログラムを合成することができた。この例題についても併せて報告する。

さらに本論文では三番目の対応付けとして証明図の正規化 (normalization) という技法を応用する。この正規化のアルゴリズムを用いると、証明図を帰納的関数あるいは原始帰納的汎関数で解釈することなく直接に実行することができる。この実行メカニズムはちょうど Prolog を一般の論理式に拡張したものに相当する。ただし、Prolog ではプログラムを実行しながら証明が行われるのに対して、正規化による計算では最初に証明を行い、出来上がった証明図を正規化する。もしデータの中に無限に長いリストが含まれている場合には、Concurrent Prolog 等の論理プログラミングにおいてはストリームとして問題なく扱われる。しかし証明図の正規化においては無限の要素に論理的な意味を与えない限り証明図が構成できず、したがって計算に移れない。本論文の第 4 章では、無限の要素に意味を与えるために、**超準解析** (non-standard analysis) を応用して論理体系を拡張した。この結果、Concurrent Prolog の動作に相当する計算を正規化によって実行することが可能になった。

第 5 章では、上述の無限リストの応用の一つとしてプログラムの同値性の判定を論じる。ここで対象とするプログラムは car, cdr, cons からなる Lisp の部分言語である (Formal Lisp と名付けた)。我々は Formal Lisp で書かれたプログラムの実行結果をトレース

の形式で表す。このトレースはちょうど証明図のような形をしている。二つのプログラムが同値であることを示すには、二つのプログラムに同じ入力を与えてトレースを作り、その二つのトレースの証明図に正規化の処理を施して同型になるか否かを判定すれば良い。この同値性の判定法は極めて簡単であるが、同値性を広範に論じている Boyer と Moore の教科書 [4] に載っている最初の例題の幾つかを解くことができる。

トレースを用いる手法の欠点として、一つのプログラムが生成するトレースは一般に無限個存在する。つまりプログラムはトレースの無限集合によって表現される。無限集合は取り扱いにくいため、トレースの集合を有限個のトレースで代表させたい。本研究では次のような方法を考案した。すなわち、プログラムが一様 (uniform) という性質を満たす時には、無限個のトレースの代わりに一つのトレースで代用することができる。ただし、そのトレースはプログラムの入力として無限リストを与えた時に得られるものである。

本論文では、古くから理論的に知られていた各種の論理的手法をコンピュータ・プログラムの世界に応用して、プログラムの合成と解析を研究した。ここで活用した理論的な成果は、コンピュータやプログラムの理論が発達する以前に得られていたものが多い。また、いずれの理論も論理学の基本的な結果として知られているものばかりである。このような理論が今日の情報工学の分野に応用できることは正に驚異であり、また先人の知恵に大いに感服する所以でもある。

1.2 本研究の背景

前節で述べたように、論理式の証明とプログラムの間には密接な関係がある。本論文は、その関係を利用してプログラムの合成と解析を論理的に研究するものである。第 2 章以降の本文中には種々の対応関係が登場するが、それらは必ずしも同一の関係ではない。第 2 章以下の各章では様々な対応関係の特徴を生かして論旨を展開する。本節では種々の対応関係に共通する基本的な事項をまとめておく。

1.2.1 プログラムと証明

コンピュータ・プログラムと論理式の証明の間の基本的な関係は、プログラムの検証 (verification) という考え方に良く現れている。プログラムの検証は Floyd[12] や Hoare[21] によって研究された。ここでは基本的な考え方をみるために、次の様に簡略化された Algol 型のプログラミング言語を考える。プログラムの構成要素はステートメントと呼ばれる。最も基本的なステートメントは代入ステートメントであり、次の形をしている。

代入ステートメント: $y \leftarrow f(x, y)$

ここに x, y は変数であり $f(x, y)$ は式である。また有限個のステートメントをセミコロンで区切って並べると、ステートメントを逐次実行することができる。

逐次実行: $S_1\ ;\ S_2\ ;\ \cdots\ ;\ S_n$

さらに二つのステートメント S_1 と S_2 を用いて次の様な条件ステートメントを作ることもできる。

条件ステートメント: $\mathbf{if}\ t(x, y)\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$

ここに $t(x, y)$ はテストと呼ばれる式である。任意のステートメント S とテスト $t(x, y)$ を用いて次の様な while ステートメントを作ることもできる。

while ステートメント: $\mathbf{while}\ t(x, y)\ \mathbf{do}\ S$

実際にプログラムの検証を進めるためには、以上の他にステートメントの列を begin と end で囲むようにしたり、START, HALT という特別なステートメントを用意することもある。ここでは詳細は省略する。

さて、以上のような構成要素を持つプログラムの検証は次のようにして行なう。まず下に示す新しい表記を導入する。

$$p(x,y)\ S\ q(x,y)$$

ここに S は検証の対象となるプログラムであり、$p(x,y)$ は前件 (precondition) と呼ばれる論理式、$q(x,y)$ は後件 (postcondition) と呼ばれる論理式である。前件および後件は条件を表しているので、上の表記は「プログラム S の実行前に条件 $p(x,y)$ が成り立っているならば、プログラムの実行後には $q(x,y)$ が成り立つ」という意味を持つ。

さて検証の目的は $p(x,y)\ S\ q(x,y)$ という主張を証明することである。$p(x,y)\ S\ q(x,y)$ という表現の中には論理式とプログラムとが同時に含まれているから、この表現を証明するための推論規則は論理式だけを扱う推論規則とは多少異なる。

プログラム S の構成要素はステートメントであるから、推論規則をステートメント毎に用意して、それらを組み合わせてプログラムに対する証明を構成する。図 1.4にはプログラムの検証に用いる推論規則をまとめておいた。

$$\{p(x,g(x,y))\}\ y \leftarrow g(x,y)\ \{p(x,y)\}$$

$$\frac{\{p\}\ S_1\ \{q\} \qquad \{q\}\ S_2\ \{r\}}{\{p\}\ S_1\ ;\ S_2\ \{r\}}$$

$$\frac{\{p \wedge t\}\ S_1\ \{q\} \qquad \{p \wedge \neg t\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ t\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{q\}}$$

$$\frac{\{p \wedge t\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ t\ \mathbf{do}\ S\ \{p \wedge \neg t\}}$$

図 1.4: プログラムの検証のための推論規則

図 1.4の最初の行は代入ステートメント $y \leftarrow g(x,y)$ に対する推論規則である。これは正確に言えば公理の形をしており、単一の表現 $\{p(x,g(x,y))\}\ y \leftarrow g(x,y)\ \{p(x,y)\}$ である。この公理は Manna の教科書 [30] の教科書にあるようにステートメントの実行後の変数 y を y' で示すと $\{p(x,g(x,y))\}\ y \leftarrow g(x,y)\ \{p(x,y')\}$ となる。この代入 $y \leftarrow g(x,y)$ を

等式 $y' = g(x,y)$ と解釈すれば「等しいもの同士を置き換えて良い」という等号 (=) の公理 $p(x,g(x,y)) \wedge y' = g(x,y) \supset p(x,y')$ と同じものになる。つまり通常の論理体系の中で扱うことが可能である。

図 1.4の 2 番目以降の推論規則は各々、ステートメントの逐次実行、条件ステートメント、while ステートメントに対応している。いずれも横棒 (—) の上には推論規則の前提、横棒の下には結論が示されている。逐次実行 S_1 ; S_2 に対する推論規則は論理式における三段論法の推論規則に良く似た形をしている。類似性の正確な議論をするためには論理体系の側を厳密に定めなければならないが、結論を言えば表 1.2の様な対応が可能であることが知られている。

3 番目の条件ステートメントに対する推論規則は、テスト t について t が成り立つ場合と $\neg t$ の場合に分けて証明をすることを示している。これは論理式の規則で言えば場合分けの証明 (case analysis) と呼ばれているものである。

推論規則	プログラム
三段論法	逐次実行
場合分け	条件分岐 (if)
数学的帰納法	ループ

表 1.2: 推論規則とプログラムとの対応

最後の while ステートメントの規則は条件ステートメントの場合に似ている。この規則の外観だけでは表 1.2に掲げられているような数学的帰納法との対応は付きにくい。しかし、while などのループに対応する論理的な構造は数学的帰納法によって表されることが知られている。

1.2.2 定理証明とプログラムの合成

論理式の世界には定理の自動証明という技術がある。特に一階述語論理に対しては導出法 (resolution method) という定理証明法が完全 (complete) であることが示されている。ここに完全な定理証明法とは、証明可能な論理式に対しては必ず証明を生成できる方

法を言う。例えば Chang と Lee[6] は導出法の優れた教科書である。この [6] の第 11 章では、定理の証明に基づく質問応答システム (Question Answering System) が論じられている。表 1.3は必要とされる応答の程度によって問題を階層的に分類したものである。分類 A~D は文献 [6] による。また分類 E はプログラムとの対応関係の説明をするために追加したものである (後藤 [69])。

分類	質問応答のクラス	証明過程からの情報抽出
A	Yes/No で答えれば良いもの	証明の成功 / 不成功が Yes/No という答に対応する。
B	Where, Who, What の付いた質問	結論に含まれる変数に代入される定数を答える。
C	How に対して一連の動作を答えるもの	結論に含まれる変数に代入される項を答える。
D	答に判断、条件分岐を含むもの	場合分けの証明 (case analysis) をトレースして条件を得る。
E	ループ、反復実行を含むもの	数学的帰納法の推論規則をトレースして反復されるものを得る。

表 1.3: 質問応答システムの分類

具体的な様子を見るために、分類 C に属する問題の代表例を次に示す。これは人工知能の分野で monkey banana problem と呼ばれている有名な問題である。

問題の状況:
猿が部屋の中にいるものと仮定する。猿はバナナを食べたいと思っているが、バナナは天井からぶら下がっている。猿は背が低くて手を伸ばしてもバナナには届かない。しかし猿は部屋の中を歩くことができるし、離れたところにある椅子を運ぶこともできる。猿はどうすればバナナを取ることができるか。

このような問題は導出法によって容易に解くことができる。答は一連の動作「椅子のところまで歩き、椅子を持ってバナナの直下まで歩き、椅子の上に乗って手を伸ばす」である。答の導出は文献 [6] を参照されたい。

上の例における猿 (むしろロボットと呼ぶ方が適切かも知れない) の動作は、まさにコンピュータの命令の実行と同じものであるから、プログラムの合成が同様の手法で実現できる。すなわち、求めるプログラムが答として出てくるような質問を仕組めば良いのである。ただし、プログラムの合成に導出法を応用するに当たっては注意しなければならない点がある。それは、導出法が古典論理に基づいていることである。その結果として、得られた答がプログラムとしては無意味になる可能性がある。(2.1節参照)

なお実際にプログラムの合成を行う際に、表 1.3の分類 C のレベルに留まっていたのでは自明なプログラム (直線状に順次実行するだけ) しか得られず、実用性に乏しい。分類 D に属するプログラムの合成例については文献 [6] に簡単な例が報告されている。なお分類 E に属するプログラムを合成するためには数学的帰納法が必要である。導出法は数学的帰納法を扱うことができないから、実際の合成システムでは導出法に加えて何らかの帰納法的な処理が必要となる。ここでの対応関係は前出の表 1.2と全く同様である。

1.2.3 証明図の正規化と計算

証明図の正規化は定理証明の技術とは趣を異にする。すなわち定理証明では与えられた論理式の証明図を作ることが目的であるのに対して、正規化では出来上がった証明図を対象として操作を行う。

正規化という操作の内容は第4章で正確に述べるが、ここでは正規化が一種の計算機構を実現していることを説明しておこう。証明図を正規化するということは証明図の中の**冗長** (redundant) な部分を除去することである。冗長な証明には何種類かの形があり、各々の形に応じた**リダクション規則** (reduction rule) を適用することにより該当する冗長な部分を除去できる。冗長な部分を含まない証明図を**正規** (normal) であると言う。普通に考えると、教科書に載っているような証明図には冗長な部分が存在するとは思われない。しかし正規である二つの証明図を合成して一つの証明図を作ると、新しい証明図の中に冗長な部分が現れることが多い。

以下では最も簡単なリダクション規則を示し、計算機構との関連を見る。論理式の証明図は幾つかの推論規則から組み立てられる。最も簡単な推論規則は三段論法と呼ばれるもので、論理式 A と論理式 A⊃B から論理式 B を導く。ここでは推論規則を表すために自然演繹法 (natural deduction) を用いることとし、推論の前提となる論理式を横棒(—)の上に書き、結論を横棒の下に書く。なお自然演繹法では三段論法に ⊃E (⊃ Elimination) という名前を付けるが、これは推論規則によって論理記号 ⊃ が除去 (Eliminate) される

からである。

$$\frac{A \quad A\supset B}{B}\supset E$$

次に示す ⊃I は ⊃E と対になる推論規則である。⊃I は論理記号 ⊃ を導入 (Introduce) する。この ⊃I は次のような推論を行なう。まず論理式 A を仮定する。その上で証明を行ない、論理式 B が導かれたとする。この時 ⊃I の規則を適用すれば論理式 A⊃B を導くことができる。なお細かいことを言えば、途中の結論 B は仮定 A に依存 (depend) するが、最終的な結論の中には A が取り込まれているから結論 A⊃B は仮定 A には依存しない。この事実を表すために仮定 A をカギカッコ [ ] で囲み、仮定 A が推論規則 ⊃I で落とされた (discharged) と言う。(仮定を落とすことの詳細は第 4 章参照。)

$$\frac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A\supset B}\supset I$$

以上の二つの推論規則を用いると冗長な証明図を作ることができる。下に示す証明図の中では ⊃I の規則の直後に ⊃E の規則が適用されている。

$$\frac{\begin{array}{c}\Pi_2\\ A\end{array} \quad \frac{\begin{array}{c}[A]\\ \Pi_1\\ B\end{array}}{A\supset B}\supset I}{B}\supset E$$

上の証明図は冗長である。すなわち部分証明 Π_1 では A を仮定して B を導き、その B に対して ⊃I を適用して結論 A⊃B を得ているのであるが、そこで導入された論理記号 ⊃ は直ちに ⊃E で除去されて最終的な結論は B となっている。

この証明を次のように変形することができる。すなわち部分証明 Π_1 の仮定 A のところに部分証明 Π_2 の結論 A を入れる。このように二つの証明 Π_1 と Π_2 を縦に接続しても全体の結論は B のまま不変である。また全体として証明となっていることにも変わりがない。しかし、使用されている推論規則の数は ⊃I と ⊃E の分だけ二つ減っている。

$$\begin{array}{c}\Pi_2\\ A\\ \Pi_1\\ B\end{array}$$

この例のように、自然演繹法の証明において論理記号を導入した直後にその論理記号を除去すると冗長な証明図となる。⊃ に関するリダクション規則は「連続した ⊃I と ⊃E の適用を省いて証明図を簡単にすること」である。⊃ 以外の論理記号のリダクション規則

は第 4 章に述べてある。

さて、⊃ に関するリダクション規則を別の見方で眺めることができる。すなわち論理式の証明図をタイプ付きラムダ式で解釈すると、A⊃B という結論を持つ証明図は $\lambda a.b$ という式に相当する。この式は次の様に読める。A の証明が a、B の証明が b である時、A⊃B の証明は $\lambda a.b$ つまり a を与えれば b を出力するものである。

この解釈で上の ⊃ のリダクションを記述すると、冗長性を持つ最初の証明は

$$(\lambda a.b)a$$

となる。ここでは証明 Π_2 が a に相当する。またリダクション後の証明図は単に B を結論とするものであるから、

$$b$$

である。このリダクションは $(\lambda a.b)a \ \rhd\ b$ と書ける。これはラムダ式の分野で β リダクションと呼ばれているものと全く同じものである (Barendregt[1])。ラムダ式における β リダクションは基本的な計算のメカニズムであるから、証明図の正規化が計算機構を実現していることの説明の一端になっている。

この他の論理記号のリダクションも同様にラムダ式のリダクションとして解釈できることが知られている (Goad[15], Howard[22], Lambek と Scott[25])。

第 2 章

直観主義論理に基づくプログラム理論

序論では古典論理の問題点を二つ指摘した。本章では直観主義論理の立場から問題を見直して解決をはかる。直観主義論理は**構成的論理** (constructive logic) の一種である。具体的な論理式を取り扱う前に構成的論理の特徴を概観しておく。構成的論理の定義には幾つかの流儀がある。Beeson の本 [2] によれば、構成的数学には少なくとも五つの異なる哲学的な立場があると言う。ただし基本的な考え方には共通点があるから、ここでは概略を紹介する。

2.1 古典論理と構成的論理

普通に論理と言えば**古典論理** (classical logic) を指す。構成的論理と古典論理との最大の相違点は $\exists x A(x)$ という形の論理式の解釈にある。この論理式は「$A(x)$ を満たすような x が存在する」という主張をしている。いわゆる存在定理の形である。

($\exists$1) 古典論理においては、$\exists x A(x)$ が主張しているのは存在自体であって x の具体的な形ではない。したがって x に相当する具体的なものに関する情報は何もない。

($\exists$2) 構成的論理において $\exists x A(x)$ を主張する時には、必ず x に相当するものを見つけるための具体的な方法を伴わなくてはいけない。この情報は必ずしも陽に x の形を述べる必要はないが、何らかの方法で x に相当する具体的な値を捜せないといけない。

上に述べた構成的論理の性質はコンピュータ・プログラムと関係が深い。プログラムの仕様 (specification) は次のような論理式で表現できる。

$$\forall x \exists z P(x, z)$$

ここに x は入力を表す変数である。z は出力を表す変数である。$P(x,z)$ はプログラムの入力 x と出力 z との間の関係 P を表している。この論理式は「すべての x に対して、適当な z が存在して $P(x,z)$ を満たす」ことを主張している。例えば、次の論理式は割算のプログラムの仕様を表す論理式である。

$$\forall x_1 x_2 \exists z_1 z_2 \{x_2 \neq 0 \supset (x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}$$

この論理式は x_1 を x_2 で割った商が z_1 で剰余が z_2 となることを表している。プログラムの仕様として見れば入力が x_1, x_2 で出力が z_1, z_2 となる。

仕様を表す論理式が構成的論理において成り立つとすると、すべての x(上の例では x_1, x_2) にたいして具体的な z(上の例では z_1, z_2) の値を求める方法が存在するはずである。ところで、x に対して z の値を求めるのはプログラムの役割に他ならない。この考察から次のことが言える。

(1) 人間がプログラムを書くということは、x に対して z を求める方法を具体的に構成していることになる。これは $\forall x \exists z P(x,z)$ という論理式を証明していることにほぼ等しい。これがプログラミングという知的作業の論理的な意味である。

(2) プログラムを書く前に、その仕様が論理式で $\forall x \exists z P(x,z)$ の形に書けたとする。もしその論理式が構成的論理の中で証明できるならば、その証明の中にプログラムに相当する情報が付随している。その情報を抽出すればプログラムを合成することができる。

以下本章では (1) の立場でプログラミングをとらえ、(2) の情報の抽出のための理論を考察する。なお本研究では、構成的論理の一つである**直観主義論理** (intuitionistic logic) を多用する。直観主義論理は構成的論理の一種類であるから、上に見た構成的論理の特徴を引き継いでいる。ただし直観主義論理の独自の特徴もあるから、簡単に説明しておく。直観主義論理は古典論理から**排中律** (law of the excluded middle) を取り除いた論理体系として規定される。排中律というのは $A \vee (\neg A)$ の形の論理式であり、古典論理においては排中律が常に成り立つ。直観主義論理において排中律を認めない理由は、$\vee$ という論理記号の解釈にある。

(V1) 古典論理においては、$A \vee B$ という論理式は「A または B の少なくとも一方が成り立つ」ということを意味する。本当にどちらが成り立つかは関知しない。

(V2) 直観主義論理における $A \vee B$ は、具体的にどちらが成り立つかを判定する方法を伴わないといけない。

この (V1) と (V2) の区別は、$\exists$ の区別と似ている。実は (V1), (V2) の区別は ($\exists$1), ($\exists$2) の区別の特別な場合と考えることができる。いずれにしても直観主義論理では $A \vee (\neg A)$ という論理式に具体的な判定の手続きが付随していない限り、天下り的に排中律を認めるわけにはいかない。この事はプログラムとして考えた方が分かり易いかもしれない。コンピュータ・プログラムにおいては、$A \vee B$ という論理式は下のような条件文に対応する。

$$\textbf{if} \quad A \quad \textbf{then} \quad \textbf{true} \quad \textbf{else} \quad B$$
$$\text{あるいは} \quad \textbf{if} \quad B \quad \textbf{then} \quad \textbf{true} \quad \textbf{else} \quad A$$

ここで肝心なのは A(または B) という条件が具体的にプログラムの中で計算できないと無意味なプログラムになってしまうことである。つまりプログラムの立場では具体的な判定が必要とされる。このプログラムとしての判定という見方は上の (V2) と同じことになる。すなわちプログラムを論理的に考察するためには直観主義論理ないしは構成的論理の立場に立つと都合が良い。

上で述べた直観主義の特徴は、序論で述べた古典論理の問題点 i) を解決するものである。すなわち、次のような例題が知られている (後藤 [67])。

例題: 整数 x を入力し、もし $x \geq 100$ ならば出力 $z = 1$ を返し、$x < 100$ ならば $z = 0$ を返すプログラムを作れ。

この時、古典論理を用いると二つの証明が可能であり、各々次の二つのプログラムが合成される。上のプログラム (1) が普通に考えられる結果であるが、下のプログラム (2) も正当な証明から得られる。

$$z \;=\; (\textbf{if} \;\; x \geq 100 \;\; \textbf{then} \;\; 1 \;\; \textbf{else} \;\; 0) \qquad (1)$$
$$z \;=\; (\textbf{if} \;\; z{=}1 \;\; \textbf{then} \;\; 1 \;\; \textbf{else} \;\; 0) \qquad (2)$$

下のプログラムは「もし出力が 1 ならば 1 を返し、さもなくば 0 を返す」という意味である。古典論理で考えれば正しいプログラムには違いないのだが、実際上は何の役にも立たない。つまり変数 z の値が 0 か 1 かを決定するためにわざわざプログラムを作ったにもかかわらず、その z の値を用いて条件分岐 (if-then-else) が組み立てられているのである。この例でも分かるように古典論理を無条件に認めると実行不可能なプログラムが合成されてしまう。そこで古典論理に基づいてプログラムの合成を考える場合には証明に制

限を加える必要がある (例えば Chang と Lee[6])。なお直観主義論理にしたがえば、上記のプログラムのうち上側のプログラム (1) に相当する証明のみが可能である。したがって合成されるのは実行可能なプログラムに限られる。

本論文では、構成的論理、特に直観主義論理を用いてプログラムの解析と合成を研究する。本研究の中では、上に述べたような論理とプログラムとの関係が大いに活用される。

2.2 古典論理の問題点

古典論理に基づいてプログラムの合成を考えると問題が生じる。この事実は以前から指摘されており、特に合成されたプログラムの実行可能性については古典論理の枠内で証明に制限を設けることにより解決がはかられてきた (Chang と Lee[6])。しかし抜本的な解決のためには、論理体系から考え直した方が良い。

本節では序論の問題点 ii) を解決する。まず問題自体を正しく認識しよう。従来のプログラム合成の研究で代表的論文と目される Manna と Waldinger[29] には次のような記述がある。(記号は原文と少し変えた)

> 『二つの論理式 $\forall x\{A(x) \supset \exists z B(x,z)\}$ と $\forall x \exists z\{A(x) \supset B(x,z)\}$ とは論理的に同値である。しかし二つの論理式は各々異なったプログラムを意味する。これは奇妙なことである。』

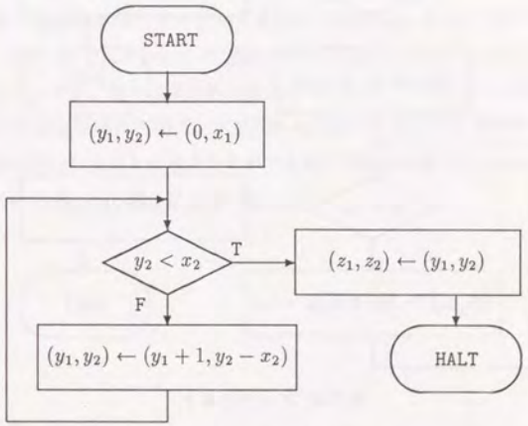

図 2.1: プログラム 1

図 2.1、図 2.2に示す二つのフローチャートは [29] から引用した割算のプログラムの例である。確かにこの二つはプログラムとしては同値ではない。すなわち、$x_2 = 0$ なる入力に対して図 2.1のプログラムは停止しないし、図 2.2のプログラムは停止する。一方、二つのプログラムに対応する論理式の方は明らかに同値である。

$$\forall x_1 \forall x_2 \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}$$
$$\equiv \forall x_1 \forall x_2 \exists z_1 \exists z_2 \{x_2 \neq 0 \supset (x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}$$

上の論理式が図 2.1に対応し、下の論理式が図 2.2に対応する。このようなプログラムと論理式との不一致は、プログラムの理論の基本的な立場「あるプログラムの持っているすべての性質はプログラムのテキスト自体からことごとく演繹される筈である」(Hoare[21]) に反するものであり、ぜひとも解決しなければならない問題である。

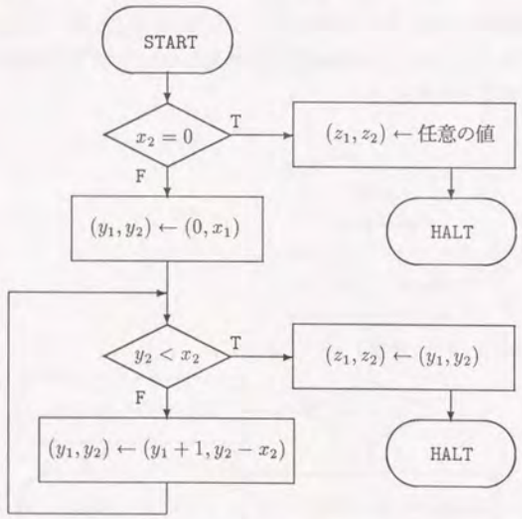

図 2.2: プログラム 2

本論文において我々のとった方法は、まず二つの論理式 $\forall x\{A(x) \supset \exists z B(x,z)\}$ と $\forall x \exists z \{A(x) \supset B(x,z)\}$ とが同値に ならない ような論理体系を捜し出し、次にその論理体系をプログラムとの関連において意味付けすることである。結論だけを先に述べると、そのような論理体系は直観主義論理であり、Kleene の recursive realizability(後述) による二つの論理式の意味は次のようになる。

$\forall x\{A(x) \supset \exists z B(x,z)\}$ … 任意の x に対して、もし $A(x)$ が真であるならば $B(x,z)$ を満たす z を出力するようなプログラム。$A(x)$ を偽とする x に対して出力 z は定義されない (undefined)。

$\forall x \exists z\{A(x) \supset B(x,z)\}$ … 任意の x に対して、必ず出力が存在する (totally defined)。$A(x)$ が真の時には $B(x,z)$ も真である。$A(x)$ が偽の時には z の値についての条件は無い。

これは丁度 Manna と Waldinger の論文 [29] で informal に主張されていることを形式的に裏付けたことに相当する。

ここで我々が用いた直観主義論理および recursive realizability は理論的には古くから知られていた概念である。また realizability とプログラムの合成の関連も Constable[8] に概要が述べられている。本論文も理論的な枠組みは [8] に示唆されているものに沿っている。ただし [8] では realizability に関する Nelson と Kleene の定理を $\forall x\{P(x) \supset \exists! y F(x,y)\}$ の形[†]についてだけ述べているのに対し、我々は一般的な形で用いたので、二つのプログラムの間の関係も論理的に論じることができた (2.4節参照)。これは [8] の明らかな拡張になっている。更に [8] では $P(x)$ が T(恒真) の時に全域的 (total) なプログラムが得られることを指摘しているが、我々は具体的な論理式 $\forall x\{A(x) \supset \exists B(x,z)\}$ と $\forall x \exists z\{A(x) \supset B(x,z)\}$ について考察を進めた結果、古典的には同値であるような変形によってもプログラムの同値性が失われることを示すことができた。詳細は次節で述べる。この結果は実際にプログラムの合成を行う場合に有用である。

† ここに論理記号 $\exists! y$ は「y が一意に存在する」ことを表す。

2.3 直観主義論理による問題の解決

前節で Manna と Waldinger の問題を説明した際に、論理的に同値 (logically equivalent) という言葉を使用した。この「同値」は正確に言うと「一階古典論理において同値」の意味である。なぜなら以下に示すように、一階直観主義 (intuitionistic) 述語論理においては先の二つの論理式は同値ではない。したがって直観主義論理に基づいてプログラムの合成を考えるならば Manna と Waldinger の問題は生じない。

次の補題 1 と補題 2 で二つの論理式の間の関係を考察する。

補題 1 一階直観主義述語論理において、
$\vdash \forall x \exists z \{A(x) \supset B(x,z)\} \supset \forall x \{A(x) \supset \exists B(x,z)\}$
が成り立つ。ここに $\vdash C$ は考えている論理体系の中で C が証明可能であることを表す。

$$\frac{A \quad B}{A \wedge B}\ \wedge\mathrm{I} \qquad \frac{A \wedge B}{A}\ \wedge\mathrm{E} \qquad \frac{A \wedge B}{B}\ \wedge\mathrm{E}$$

$$\frac{A}{A \vee B}\ \vee\mathrm{I} \qquad \frac{B}{A \vee B}\ \vee\mathrm{I} \qquad \frac{A \vee B \quad \begin{array}{c}[A]\\ \vdots \\ C\end{array} \quad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C}\ \vee\mathrm{E}$$

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \supset B}\ \supset\mathrm{I} \qquad \frac{A \quad A \supset B}{B}\ \supset\mathrm{E}$$

$$\frac{A(a)}{\forall x(A(x))}\ \forall\mathrm{I} \qquad \frac{\forall x(A(x))}{A(t)}\ \forall\mathrm{E}$$

$$\frac{A(t)}{\exists x(A(x))}\ \exists\mathrm{I} \qquad \frac{\exists x(A(x)) \quad \begin{array}{c}[A(a)]\\ \vdots \\ C\end{array}}{C}\ \exists\mathrm{E}$$

$$\frac{\curlywedge}{A}\ \curlywedge \qquad (\curlywedge \text{ は「恒に偽」を表わす})$$

図 2.3: 一階直観主義述語論理の推論規則 (自然演繹法)

補題 1 の proof[‡] を示す前に、一階直観主義述語論理の論理体系の定式化について述べて

[‡]本論文では「証明」を対象とする定理や補題を取り扱う。混乱を防ぐために定理および補題自身の「証明」を指す時には proof と書くことにする。

おこう。図 2.3には Gentzen[13] が提案した自然演繹法 (natural deduction) による一階直観主義述語論理の推論規則をまとめておいた。横棒 (—) の上に書いてある論理式が前提 (premise)、横棒の下が結論 (conclusion) である。また棒の横に書いてあるのは推論規則の名前である。推論規則の名前は次のように統一的に決められている。すなわち $\wedge$I という推論規則は $\wedge$(AND) という論理記号を導入 (Introduction) する規則である。したがって $\wedge$I 規則の結論の論理式の中には $\wedge$ が現れている。逆に $\wedge$E の規則は $\wedge$ を除去 (Eliminate) する推論規則である。したがって $\wedge$E の仮定の中には $\wedge$ が現れている。なお $\curlywedge$ の推論規則だけは、この命名法によらない。$\curlywedge$ は「恒に偽」を意味する論理記号である。なお推論規則の中には前提の論理式にカギカッコ [ ] が付いているものがある。これについては第 4 章で説明する。

自然演繹法の証明は図 2.3の推論規則を上下に重ねた木 (tree) の形を取る。このことを強調する場合には証明木 (proof tree) あるいは証明図 (proof figure) と呼ぶ。

補題 1の proof: 図 2.3の推論規則を用いて証明図 (図 2.4) を作れば良い。(proof 終)

$$\dfrac{\dfrac{\dfrac{\forall x \exists z \{A(x) \supset B(x,z)\}}{\exists z \{A(a) \supset B(a,z)\}}\ \forall\mathrm{E} \qquad \dfrac{\dfrac{\dfrac{\dfrac{[A(a) \supset B(a,c)] \qquad [A(a)]}{B(a,c)}\ \supset\mathrm{E}}{\exists z B(a,z)}\ \exists\mathrm{I}}{A(a) \supset \exists z B(a,z)}\ \supset\mathrm{I}}{\forall x \{A(x) \supset \exists z B(x,z)\}}\ \forall\mathrm{I}}{\forall x \{A(x) \supset \exists z B(x,z)\}}\ \exists\mathrm{E}}{\forall x \exists z \{A(x) \supset B(x,z)\} \supset \forall x \{A(x) \supset \exists z B(x,z)\}}\ \supset\mathrm{I}$$

図 2.4: 自然演繹法による証明図

補題 2 一階直観主義述語論理において $\forall x \{A(x) \supset \exists z B(z)\} \supset \forall x \exists z \{A \supset B(z)\}$ は成り立たない。

補題 2の proof は補題 1に比べて難しい。証明図を作る方法で示すならば「補題 2の論理式に対しては、いかなる証明図も構成できない」ことを言わなければならない。これは大変である。ここでは証明可能性と等価な概念である妥当性を用いて示す (後藤 [60])。一階直観主義述語論理における妥当性の定義は下の Kripke モデルを用いる。

Kripke モデルは下の様に作れば良い。この定義は大変抽象的なものであるから分かりにくい。すぐ後に簡単な説明を追加する。

定義 1 Kripke モデルとは順序付けられた 4 つ組 $\langle \mathcal{G}, \mathcal{R}, \models, \mathcal{P} \rangle$ のことである。但し、$\mathcal{G}$ は空でない集合、$\mathcal{R}$ は $\mathcal{G}$ の上の *transitive* かつ *reflexive* な関係 (relation)、$\models$ は $\mathcal{G}$ の要素と論理式の間の下のような条件を満たす関係、$\mathcal{P}$ は $\mathcal{G}$ から定数の集合 (空でない) への写像である。

条件: すべての $\Gamma \in \mathcal{G}$ に対して以下が成り立つ。記号 Γ^* は $\{\Delta \in \mathcal{G} \mid \Gamma\mathcal{R}\Delta\}$ の任意の元を表す。記号 $\widehat{\mathcal{P}}(\Gamma)$ は定数として $\mathcal{P}(\Gamma)$ だけを用いて構成される論理式の集合を表す。

Q0. $\mathcal{P}(\Gamma) \subseteq \mathcal{P}(\Gamma^*)$

Q1. $\Gamma \models A \Rightarrow A \in \widehat{\mathcal{P}}(\Gamma)$, ここに A は素論理式 (*atomic formula*) とする。

Q2. $\Gamma \models A \Rightarrow \Gamma^* \models A$, ここに A は素論理式

Q3. $\Gamma \models (X \wedge Y) \Leftrightarrow \Gamma \models X$ かつ $\Gamma \models Y$

Q4. $\Gamma \models (X \vee Y) \Leftrightarrow (X \vee Y) \in \widehat{\mathcal{P}}(\Gamma)$ であり、しかも $\Gamma \models X$ または $\Gamma \models Y$ の一方が成り立つ

Q5. $\Gamma \models \neg X \Leftrightarrow \neg X \in \widehat{\mathcal{P}}(\Gamma)$ であり、すべての Γ^* に対して $\Gamma^* \not\models X$

Q6. $\Gamma \models (X \supset Y) \Leftrightarrow (X \supset Y) \in \widehat{\mathcal{P}}(\Gamma)$ であり、すべての Γ^* に対してもし $\Gamma^* \models X$ ならば $\Gamma^* \models Y$

Q7. $\Gamma \models \exists x X(x) \Leftrightarrow$ 或る $a \in \mathcal{P}(\Gamma)$ に対して $\Gamma \models X(a)$

Q8. $\Gamma \models \forall x X(x) \Leftrightarrow$ すべての Γ^* およびすべての $a \in \mathcal{P}(\Gamma^*)$ に対して $\Gamma^* \models X(a)$

(定義終)

形式的な定義だけでは、いかにもわかりにくい。ここで informal な説明を補う。

Kripke モデルの説明の中で、一番分かりにくいのは最初に出てくる集合 $\mathcal{G}$ である。実は $\mathcal{G}$ の一つの元 (要素) は一つの世界 (world) を表す。古典論理のモデルにおいては世界は唯一つしか存在しないが、直観主義論理では多数の世界において同時に論理式の解釈を行う。これが直観主義論理のモデルの特徴である (Hughes と Cresswell[23], [40])。

各々の世界のことを**可能世界** (possible world) と呼ぶ。可能世界の間の関係 $\mathcal{R}$ は**到達可能関係** (accessible relation) と呼ばれる。簡単に言うと、世界 Γ_1 から世界 Γ_2 が「見える」(到達可能である) ことを $\Gamma_1\mathcal{R}\Gamma_2$ と表す。この関係 $\mathcal{R}$ は定義により transitive であるから図 2.5のように $\Gamma_1\mathcal{R}\Gamma_3$ かつ $\Gamma_3\mathcal{R}\Gamma_4$ であれば $\Gamma_1\mathcal{R}\Gamma_4$ が自動的に言える。また reflexive と規定されているから、すべての Γ に対して $\Gamma\mathcal{R}\Gamma$ が成り立つ。

各可能世界には定数 (constant) が存在する。原理的には各世界の定数は異なっていても良いが、定義によれば $\mathcal{P}(\Gamma) \subseteq \mathcal{P}(\Gamma^*)$ という条件がある。すなわち Γ から見える世界の定数には必ず Γ と同じ定数を含まなければならない。言い換えれば、Γ における論理式 $A \in \widehat{\mathcal{P}}(\Gamma)$ は必ず Γ^* でも論理式として扱われる。

このように考えると、例えば条件の Q6 は次の様な意味を持つ。

[Q6 の意味] ある世界 Γ において論理式 $(X \supset Y)$ が真であることは、次の条件と同値である。

i) 論理式 $(X \supset Y)$ は Γ の世界の論理式である。$(X \supset Y) \in \widehat{\mathcal{P}}(\Gamma)$

ii) Γ から見えるいかなる世界 Γ^* においても、もし X が Γ^* で真であるならば Y も必ず Γ^* において真である。

その他の条件も同様に解釈される。補題 2の proof の中では Q7 も使われる。

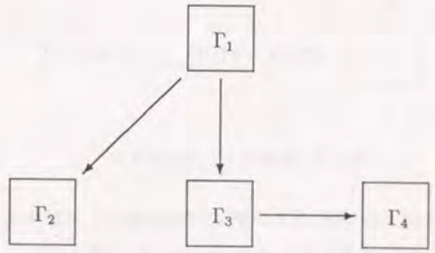

図 2.5: Kripke モデルにおける関係 $\mathcal{R}$

定義 2 ある論理式 X がモデル $\langle \mathcal{G}, \mathcal{R}, \models, \mathcal{P} \rangle$ において妥当 (*valid*) であるとは、$X \in \widehat{\mathcal{P}}(\Gamma)$ であるようなすべての $\Gamma \in \mathcal{G}$ に対して $\Gamma \models X$ が成り立つことである。どのようなモデルに対しても妥当である論理式を単に妥当 (*valid*) であると言う。

定理 1　X が直観主義述語論理で証明可能であれば X は妥当である。X が証明可能でないとすれば反例 (*counter model*) が作れる。

Proof Fitting[11] 参照。

補題 2の proof　定理 1で言うところの反例を示せば良い。ここでは Kripke モデルを用いる。話を簡単にするために $\{A \supset \exists z B(z)\} \supset \exists z\{A \supset B(z)\}$, A は z を含まない、としても一般性を失わない。

$\mathcal{G} = \{\Gamma_1, \Gamma_2\}$

$\Gamma_1 \mathcal{R} \Gamma_2$, $\Gamma_1 \mathcal{R} \Gamma_1$, $\Gamma_2 \mathcal{R} \Gamma_2$.

$\mathcal{P}(\Gamma_1) = \{\mathbf{1}\}$, $\mathcal{P}(\Gamma_2) = \{\mathbf{1}, \mathbf{2}\}$

$\Gamma_2 \models B(\mathbf{2}), A$

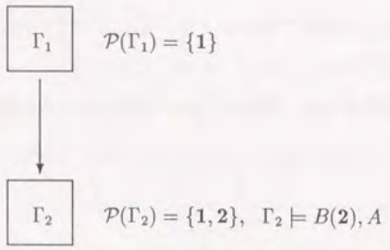

図 2.6: 簡単な Kripke モデル

これを図示すると図 2.6の様になる。図 2.6では定数として数字 $\mathbf{1}, \mathbf{2}$ を用いている。まず $\Gamma_1 \models A \supset \exists z B(z)$ を示す。$A \supset \exists z B(z) \in \widehat{P}(\Gamma_1)$ は良い。次にすべての Γ_1^* に対して $\Gamma_1^* \models A$ ならば $\Gamma_1^* \models \exists z B(z)$ であることを示す必要があるが、まず Γ_1 では $\Gamma_1 \models A$ が成り立たないから vacuous に正しい。また Γ_1^* には Γ_2 も含まれるが、Γ_2 では $a \in \mathcal{P}(\Gamma_2)$ として $\mathbf{2}$ を取れば $\Gamma_2 \models \exists z B(z)$ が成り立つから良い。

次に $\Gamma_1 \not\models \exists z\{A \supset B(z)\}$ を示す。次のことに注意する。

$\Gamma_1 \models \exists z\{A \supset B(z)\}$

$\Leftrightarrow$ 或る $a \in \mathcal{P}(\Gamma_1)$ に対して $\Gamma_1 \models A \supset B(a)$

$\Leftrightarrow$ 或る $a \in \mathcal{P}(\Gamma_1)$ に対して $A \supset B(a) \in \widehat{\mathcal{P}}(\Gamma_1)$ かつ $\forall \Gamma_1^*$, if $\Gamma_1^* \models A$ then $\Gamma_1^* \models B(a)$

最後の条件は、このモデルでは明らかに成り立たない ($\Gamma_2 \not\models B(\mathbf{1})$ であるから)。以上二つの結果をまとめると、$\Gamma_1 \models A \supset \exists z B(z)$ かつ $\Gamma_1 \not\models \exists z\{A \supset B(z)\}$ であるから図 2.6は求める反例になっている。(proof 終)

このように二つの論理式はもはや同値ではない。したがって直観主義論理を用いて Manna と Waldinger の問題を解決したことになる。

ところで、再び図 2.1, 図 2.2のプログラムを眺めてみると、この二つのプログラムは同値ではないが、良く似た構造を持っている。この両者の間の関係は拡張 (extension) と呼ばれるものである。すなわち、プログラム 1 の値が定義される時にはいつでもプログラム 2 の値も定義され、これらの値は相等しい。このような拡張関係は Manna と Cooper による二階述語論理による表現 [30] では論理記号 $\supset$ で表されるが、我々の直観主義論理の場合にも同様なことを期待しても良いであろうか。この問題を次節で扱うことにする。

2.4 論理式とプログラムとの対応関係

本節では Kleene[27] が提案した recursive realizability という考え方を用いて論理式とプログラムとの対応付けを考察する。

realizability という概念には種々のものが提案されているが (Kleene と Vesley[28])、ここでは Kleene[27] に沿って考察する。前節の議論は一階直観主義述語論理において行なったが、ここではその論理の上に作られる自然数論 (number theory) にまで話を広げる。本章では割算のように自然数の性質を用いたプログラムを扱うため、論理式の側でも自然数が扱えるように準備を整える必要があるためである。一般にプログラムの中では種々のデータ構造が扱われる。自然数は最も基本的なデータ構造の例である。図 2.7には自然数の公理と推論規則を自然演繹法で定式化する場合の追加分を掲げておく。

図 2.7には自然数論固有の 7 つの推論規則と数学的帰納法 (IND) が載っている。この図の中で t=t は公理のように使われる。また $p_1, p_2, \ldots$ は述語記号で原始帰納的関数 (primitive recursive function) のグラフを表す。関数記号 $s(\cdot)$ は successor function つまり $s(a) = a + 1$ を意味する。

$$\mathrm{t}=\mathrm{t} \qquad \frac{\mathrm{t}=\mathrm{u}}{\mathrm{u}=\mathrm{t}} \qquad \frac{\mathrm{t}=\mathrm{u} \quad \mathrm{u}=\mathrm{v}}{\mathrm{t}=\mathrm{v}} \qquad \frac{\mathrm{t}_i=\mathrm{u}_i \quad \mathrm{p}_k(\mathrm{t}_1,\ldots,\mathrm{t}_i,\ldots,\mathrm{t}_n)}{\mathrm{p}_k(\mathrm{t}_1,\ldots,\mathrm{u}_i,\ldots,\mathrm{t}_n)}$$

$$\frac{0=s(\mathrm{t})}{\curlywedge} \qquad \frac{\mathrm{t}=\mathrm{u}}{s(\mathrm{t})=s(\mathrm{u})} \qquad \frac{s(\mathrm{t})=s(\mathrm{u})}{\mathrm{t}=\mathrm{u}}$$

$$\frac{\mathrm{A}(0) \quad \begin{array}{c}[\mathrm{A}(\mathrm{a})]\\ \vdots \\ \mathrm{A}(s(\mathrm{a}))\end{array}}{\mathrm{A}(\mathrm{t})}\ \mathrm{IND}$$

図 2.7: 自然数論の推論規則 (自然演繹法)

Kleene の **realizer** という概念を一口で言えば「真 (true) である論理式には必ずその論理式を実現 (realize) する数が付随している」というものである。ここで数というのは一般には帰納的関数を表わす**ゲーデル数** (Gödel number) のことである。

定義 3

(I) 閉論理式 A が *realizable* であるとは、A を *realize* するようなゲーデル数が表 2.1にしたがって存在することである。

(II) 開論理式 $A(y_1, y_2, \ldots, y_m)$, $m \geq 0$ が *realizable* であるとは全域的な帰納的関数 φ が存在して、すべての $\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m$ に対して $\varphi(\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ が $A(\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ を *realize* することである。ここに $\bar{y}_1$ の様に変数の上に横棒を付けたものは自然数の上を動く変数を表す。

realize する数	realize される閉論理式
0 (ゼロ)	true である素論理式 (論理記号を含まない論理式)
$2^a \cdot 3^b$ 但し a が A を realize し、 かつ b が A を realize する。	$A \wedge B$
$2^0 \cdot 3^a$ 但し a が A を realize する、また は $2^1 \cdot 3^b$ 但し b が B を realize する。	$A \vee B$
部分帰納的関数 φ のゲーデル数、但し φ は A を realize するような a に対し て定義され、$\varphi(a)$ は B を realize する。	$A \supset B$
$A \supset (1=0)$ を realize するゲーデル数。	$\neg A$
$2^{\bar{x}} \cdot 3^a$ 但し a は $A(\bar{x})$ を realize する。 ($\bar{x}$ は x の値としての一つの自然数)	$\exists x A(x)$ ($A(x)$ の自由変数は x のみ)
全域的な帰納的関数 φ のゲーデル数 但し φ はすべての $\bar{x}$ に対して定義さ れ、$\varphi(\bar{x})$ が $A(\bar{x})$ を realize する。	$\forall x A(x)$ ($A(x)$ の自由変数は x のみ)

表 2.1: 閉論理式に対する realizer

上の表 2.1の中に $A \supset (1=0)$ という表現があるが、$1=0$ は $\curlywedge$(恒に偽) の意味で使われている。recursive realizability に関しては次の定理が本質的である。

定理 2 (Nelson and Kleene) 一階直観主義述語論理の上の自然数論において $\Gamma \vdash A$ である時、前提となる論理式の集合 Γ に含まれる論理式がすべて *realizable* であれば、A も必ず *realizable* である。

Proof 省略。Kleene[27] 参照。

上の定理の応用を考える。なお以下の応用ではすべてΓが空の場合を扱う。今、一階直観主義述語論理の上の自然数論で$\vdash A \supset B$が言えたとする。定義によれば、これは部分帰納的関数φの存在を意味する。但しφはaがAをrealizeする時、$\varphi(a)$がBをrealizeするようなものである。我々はこのφを

$$\lambda a.b$$

と書く。ここに$b = \varphi(a)$である。またφのゲーデル数を表わすためにλをΛに変えて次の様に書くことにする。

$$\Lambda a.b$$

この記法を用いることにより、ある論理式が与えられた時にそれをrealizeする数を簡潔に表現できる。

例: $A \supset (B \supset A \wedge B)$という論理式をrealizeする数は$\Lambda a \Lambda b.(2^a \cdot 3^b)$である。

ここでは次の点に注意すべきである。第一に$\lambda a.b$は$\lambda a.b(a)$とは異なるものである。すなわちb自体が最初から単独で存在するのではなくて、あくまでも$\lambda a.b$という関数として存在するもので、aが無ければbは存在しない。そのaが存在するためには$\vdash A$ならば十分である、という意味において部分帰納的関数なのである。第二にゲーデル数としての$\Lambda a.b$の存在は数aの存在に無関係に言えることである。つまりaはラムダ(Λ)の束縛変数(bounded variable)に過ぎず、$\Lambda a.b$は関数(に相当するゲーデル数)である。

Aがrealizableであること、すなわちaが存在すること、について我々は次のテーゼを受け入れることにしよう。

> **Kleeneの Thesis:** realizableであることは直観主義的な「真」(true)を意味する。

直観主義的な真という言葉の詳しい説明はFitting[11], KleeneとVesley[28], 松本[36]などを参照されたい。本節では以後「真」(true)、「偽」(false)、あるいは「成り立つ」「満たす」等の用語をこの意味で用いることにする。

定理 3 一階直観主義述語論理の上の自然数論において、

(a) $\vdash \forall x\{A(x) \supset \exists z B(x,z)\}$ならば、次の様な部分帰納的関数$\varphi$が存在する。すなわち$\varphi$の定義域は$\{\bar{x} \mid A(\bar{x})\}$, $\varphi(\bar{x})$は次の性質を満たす: $B(\bar{x}, \varphi(\bar{x}))$.

(b) $\vdash \forall x \exists z\{A(x) \supset B(x,z)\}$ならば、次の様な全域的帰納的関数$\psi$が存在する。すなわち$\psi$の定義域はすべての自然数、$\psi(\bar{x})$は次の性質を満たす: $A(\bar{x}) \supset B(\bar{x}, \psi(\bar{x}))$.

(c) 上の(b)と同じ条件が満たされる時、次の様な部分帰納的関数σが存在する。すなわちσの定義域は(b)で定まるψを用いて$\{\psi(\bar{x}) \mid \bar{x}$は$A(\bar{x})$を満たす$\}$. σは次の性質を満たす: 上の(a)で定めたφに対して$\forall x\{\varphi(\bar{x}) \simeq \sigma(\psi(\bar{x}))\}$. ここに$\simeq$は部分帰納的関数としての等号、すなわち両辺ともに定義されない時(*undefined*)にも等しいとする等号である。

Proof 付録A.1を参照されたい。ここでは結果を簡単に表2.2にまとめておく。(proof終)

論理式	realizeする数	性質
$\forall x\{A(x) \supset \exists z B(x,z)\}$	$e = \Lambda\bar{x}\Lambda a.(2^{\bar{x}} \cdot 3^b)$	部分帰納的関数φが存在して $B(\bar{x}, \varphi(\bar{x}))$を満たす。
$\forall x \exists z\{A(x) \supset B(x,z)\}$	$f = \Lambda\bar{x}.(2^{\bar{x}} \cdot 3^{\Lambda a.b})$	全域的帰納的関数ψが存在して $A(\bar{x}) \supset B(\bar{x}, \psi(\bar{x}))$を満たす。
$\forall x \exists z\{A(x) \supset B(x,z)\}$ $\supset \forall x\{A(x) \supset \exists z B(x,z)\}$	$g = \Lambda p \Lambda\bar{x}\Lambda a.$ $(2^{\{p(\bar{x})\}_0} \cdot 3^{\{p(\bar{x})\}_1(a)})$	部分帰納的関数σが存在して $\forall x\{\varphi(\bar{x}) \simeq \sigma(\psi(\bar{x}))\}$を満たす。

表 2.2: realizeする数のまとめ

表2.2の結果から次の様なことが言える。

1. 論理式$\forall x\{A(x) \supset \exists z B(x,z)\}$は2.2節のプログラム1に対応する。この論理式は部分帰納的関数に相当するものである。したがってプログラム1において$x_2 = 0$の時に出力がundefinedになるのは当然である。

2. 論理式$\forall x \exists z\{A(x) \supset B(x,z)\}$は2.2節のプログラム2に対応する。この論理式は全域的帰納的関数に相当するものである。したがってプログラム2においては$x_2 = 0$の場合でも出力は存在する。

3. 二つのプログラムの間の関係は論理式$\forall x \exists z\{A(x) \supset B(x,z)\} \supset \forall x\{A(x) \supset \exists z B(x,z)\}$で表される。この論理式自体は部分帰納的関数$\sigma$に相当する。関数$\sigma$はプログラム2の出力をプログラム1の出力に変換する働きを持つ。すなわち、拡張(extension)の関係にある二つのプログラムの間の変換プログラムの役割を果たす。

このように定理3の結果を用いて論理式と帰納的関数の関係を明確に述べることができた。なおσはプログラム2の出力をプログラム1の出力にundefinedの時も含めて変換するようなプログラムを意味する。本章で示した様に直観主義論理を用いると、論理式とプログラムの対応がうまく説明できる。次章では、このような対応関係を用いて実際にプログラムを合成する方法を考察する。

第 3 章

直観主義論理によるプログラムの合成法

本章では1958年に論理学者Gödelが考案した論理式の解釈法[16]（ゲーデル解釈あるいは掲載された雑誌名を取ってDialectica解釈とも言う）を活用して、論理式をプログラムの世界に対応付ける。ここでの主眼はプログラムを合成することにある。そのためには個々の論理式の解釈を積み重ねて証明図全体の解釈を得ることが重要になる。我々は、ここで示した方法を実際にLispの処理系を用いて実装し、プログラム合成の実験を行なった(後藤[62], Goto[64], [65])。

3.1 関数によるプログラムの表現

ゲーデル解釈の正確な定義はすぐ後で述べることにして、最初に話の筋道を簡単に説明しておく。今、プログラムの仕様を表す論理式が$\forall x \exists z A(x,z)$の形をしていると仮定する。この論理式が意味を持つためには、論理体系を定めなければならない。ここでは直観主義論理に基づく自然数論の中で考える。この論理体系は前章と全く同じものである。この体系のことをHeyting Arithmetic、略してHAなどとも呼ぶ。Heytingは論理学者の名前である。さて上の論理式の意味は、任意のxに対して或るzが存在して$A(x,z)$が成り立つというものである。このzはxが与えられる度にxに応じて定まれば良い。そこでxに対するzの対応関係を関数fを使って$z = f(x)$と表すことにする。

$$f : x \mapsto z$$

このfを強調して論理式を書直すと、$\exists f \forall x A(x, f(x))$という論理式を得る。束縛変数の順序が元の論理式では$\forall x \exists z$であったものが、fを使った方では$\exists f \forall x$と反転していること

とに注意して頂きたい。変数 z は変数 x に依存するが、関数 f 自体は変数 x の値によらず一様に存在するからである。

このような関数 f は Skolem 関数と呼ばれるものと同等の働きをする。ただし、Skolem 関数は限量記号を除去するために導入されるものであるから、以下で問題とする $\exists f$ という表現は現れない。さて新しい論理式 $\exists f \forall x A(x, f(x))$ を巡って次の二つの問題が発生する。

問題 1 関数 f の具体的な形を求めることができるか。できるとすれば、どのような方法を使えば求まるのか。

問題 2 自然数論において $\exists f \forall x A(x, f(x))$ という表現が許されるか。また許されないとすれば、どのような論理体系を想定すれば良いか。

先回りして解答を述べておくと、問題 1 は Yes であり、その方法は $\forall x \exists z A(x, z)$ の証明図に対してゲーデル解釈という処理を施せば良い。これが本章の主題である。また問題 2 は No であり、$\exists f$ という表現は関数変数を $\exists$ で束縛しているから一階述語論理の範囲を逸脱している。つまり、一階述語論理に基づく自然数論 (HA) で許される変数は対象変数のみであるから、関数変数 f を用いるためには論理体系を再構成しなければならない。

そこで、次節では $\exists f$ という表現を許す論理体系を考察する。ゲーデル解釈はその論理体系の上で意味を持つ。なお本章では第 2 章と本質的には同じ論理体系を用いるが、定式化には自然演繹法を用いない。本章で採用した公理と推論規則の選び方は Gödel の原論文 [16] に基づくもので、彼の解釈を記述するのに便利な体系になっている。図 3.1~3.3に本章で用いる HA(直観主義自然数論) の公理と推論規則をまとめておく。まず図 3.1には命題論理の部分すなわち変数を含まない論理式に関する公理と推論規則を掲げた。

図 3.1の公理の中には ⅄ という記号が現れている。これは第 2 章と同じように「恒に偽」を表す。なお ⅄ の記号と ¬ とは密接な関連がある。すなわち ¬A は (A⊃⅄) と同値であることが ¬A⊃(A⊃⅄) および (A⊃⅄)⊃¬A から分かる。

$$\frac{A \quad A\supset B}{B} \qquad \frac{A\supset B \quad B\supset C}{A\supset C}$$

$$\frac{(A\wedge B)\supset C}{A\supset(B\supset C)} \qquad \frac{A\supset(B\supset C)}{(A\wedge B)\supset C}$$

$$\curlywedge\supset A \qquad \neg A\supset(A\supset\curlywedge) \qquad (A\supset\curlywedge)\supset\neg A$$

$$(A\vee A)\supset A \qquad A\supset(A\wedge A)$$

$$A\supset(A\vee B) \qquad (A\wedge B)\supset A$$

$$(A\vee B)\supset(B\vee A) \qquad (A\wedge B)\supset(B\wedge A)$$

$$\frac{A\supset B}{(C\vee A)\supset(C\vee B)}$$

図 3.1: 直観主義自然数論の公理と推論規則 (命題論理の部分)

図 3.2には述語論理の部分すなわち変数と限量記号 (∀, ∃) に関する公理と推論規則をまとめた。ここに (*) の推論規則においては「B は自由変数 x を含まない」、また (†) の推論規則に関しては「項 t は A(t) の x の位置に代入可能 (free for x in A(x)) である」という条件が課せられている。

$$\frac{B\supset A(x)}{B\supset\forall xA(x)}\ (*) \qquad \forall xA(x)\supset A(t) \quad (\dagger)$$

$$A(t)\supset\exists xA(x) \qquad \frac{A(x)\supset B}{\exists xA(x)\supset B}\ (*)$$

図 3.2: 直観主義自然数論の公理と推論規則 (述語論理の部分)

$x=x$

$(x=y) \supset (y=x)$

$(x=y \wedge z=y) \supset x=z$

$x_i=x_i' \supset \varphi(x_1,\ldots,x_i,\ldots,x_n) = \varphi(x_1,\ldots,x_i',\ldots,x_n)$

但し φ は任意の n 変数の関数記号, $1 \leq i \leq n$.

$s(x) \neq 0$, s は successor function

$s(x)=s(y) \supset x=y$

$$\frac{A(0) \quad \forall x(A(x) \supset A(s(x)))}{\forall x A(x)} \quad \text{(数学的帰納法)}$$

図 3.3: 自然数論に固有の公理と推論規則

最後に図 3.3には自然数論に関する固有の公理と推論規則とがまとめてある。これは第2章の図 2.7と同じ内容を表している。なお厳密に言えば、図 3.3には原始帰納的関数を定義する公理 (defining axioms) を含ませるべきであるが、煩雑になるので省略する。自然数論に固有の公理の中には等号 (=) を扱う公理が含まれていることに注意しておこう。

3.2 有限のタイプの汎関数

本節ではゲーデル解釈の対象となる有限のタイプ (finite type) の汎関数を紹介する。ここでタイプという用語はプログラミング言語におけるデータ・タイプ (data type) という言葉とほぼ同義に使われている。ただし本節で登場するタイプは、自然数を表わすタイプを基本として組み立てられているため、プログラミングにおける整数 (integer) というタイプよりも狭い範囲を取り扱っている。

前節で登場した $\exists f \forall x A(x, f(x))$ という論理式について考察を続ける。この論理式の難点は $\exists f$ という表現にある。すなわち関数変数を $\exists$ で束縛しているために、この論理式を一階述語論理の上の自然数論で扱うことができない。そこで、関数 f を含むような論理体系を考察する。まず f は変数 x から変数 z への対応関係を表すものであった。両変数ともに自然数を表している。したがって関数 f は自然数から自然数への関数である。このような変数および関数の種別をコンピュータ・プログラムの世界ではデータ・タイプと呼んでいる。ここでも同様にタイプを考える。

$$\begin{array}{llll} f: & x & \mapsto & z & \text{(対応関係)} \\ f: & \text{自然数} & \rightarrow & \text{自然数} & \text{(タイプ)} \end{array}$$

以下では簡単のために自然数のタイプを単に o と書くことにする。この o という記法を用いると同じ内容が下のように書ける。

$$f: \ \mathrm{o} \rightarrow \mathrm{o} \quad \text{(タイプ)}$$

このように関数 f は o(自然数) というタイプから o(自然数) というタイプへの関数である。この事実を関数 f のタイプが o→o であると言う。タイプが o→o になる具体的な関数はいろいろ考えられる。

例えばもっとも基本的な関数 $s(\cdot)$, すなわち successor function, $s(x) = x+1$ のタイプは $\mathrm{o} \rightarrow \mathrm{o}$ である。なお $s(x)$ の右辺に $x+1$ という表現を使ったので、s を加算 "+" を用いて定義したかのように見えるかも知れない。この説明は便宜的なものであり、s の方が本来 + よりも基本的な関数である。ちょうど s はコンピュータの機械語のインクリメント命令に相当する関数である。図 2.7, 図 3.3においても s が公理と推論規則の中に登場していたことを思い出そう。

次にやや人工的な例であるが、x に対して $x+5$ を返す関数 $f(x) = x+5$ などもタイプ o→o の関数である。もっともタイプが o→o にならない関数も具体的に考えられる。

例えば上に出た加算の関数 $x+y$ は2引数の関数である。x も y も自然数つまりタイプ o であるとして、この "+" のタイプを考える。このような2引数の関数に対しては、1引数の関数を反復して用いれば良い。答を先に言うと $x+y$ のタイプは次のようになる。

$$\mathrm{o}\to(\mathrm{o}\to\mathrm{o})$$

これは、次のような考え方である。上の例で見たように $1+y, 2+y, 3+y, \ldots$ などは皆タイプ o→o の関数である。関数 $x+y$ の x を一つだけを取り出して考えてみると、x に1を与えると $1+y$ が返る、x に2を与えると $2+y$ が返る、と考えれば1引数の関数と見なすことができる。つまり自然数 (x) を一つ与えるとタイプ o→o の関数が得られるのである。

タイプ o		タイプ o→o
1	$\mapsto$	$1+y$
x	$\mapsto$	$x+y$

これはタイプ o→(o→o) と表わすことができる。このように考えると、タイプは o→o に限らず、多くのタイプがあることが分かる。なお以下では x, z のように自然数を表す変数をゼロ引数の関数と見なすことにする。

x: タイプ o の関数
z: タイプ o の関数

このような関数を全部扱うために次の定義を行う。なお下の定義の中の**汎関数** (functional) という言葉は「関数の関数」を意味する言葉である。

定義 4 有限のタイプの汎関数 (*functional*)

- タイプ o の汎関数とは自然数のことである。
- タイプ $\tau\to\sigma$ の汎関数とはタイプ τ の汎関数にタイプ σ の汎関数を対応させる規則のことである。

結局有限のタイプの汎関数とは、自然数、自然数の上の関数、その関数の上の汎関数、を一緒にしたものである。つまり極めて一般的な形をしているのであるが、本章で用いるのはそのうち原始帰納的な (primitive recursive) 汎関数と呼ばれる族である。定義5参照。なお有限のタイプと言うからには無限のタイプというものもある筈である。その推測は正しい。例えばタイプを表す変数を導入した体系は、事実上無限のタイプを持つことに相当する。それに比べると、本節で取り扱うタイプは自然数に相当するタイプ o を多段に重ねた構造をしており、一見複雑ではあってもタイプ定数しか現れない。(タイプの上を動く変数の使用例は本論文の付録 A.3にある。)

次の定義には論理体系 TT がまとめられている。TT は HA を有限のタイプの上に拡張したものである。

定義 5 論理体系 TT (Typed Terms)

- 記号 o はタイプである。もし σ と τ とがタイプであれば $\tau\to\sigma$ もタイプである。
- 論理体系 TT の言語は次のようなものを含む。
  対象定数 (*individual constant*) 0, s (*successor function*), $=$ (等号),
  各タイプの対象変数 (自由変数および束縛変数),
  o, $\to$, $\rho[\,]$, λ(ラムダ),
  論理記号 $\neg, \wedge, \vee, \supset$.
- TT の項 (term) および原始帰納的汎関数 (*primitive recursive functional*)

  1) 対象定数 0 はタイプ o の項である。

  2) タイプ τ の自由変数はタイプ τ の項である。

  3) f がタイプ o の項ならば、$s(f)$ はタイプ o の項である。

  4) f がタイプ σ の項、x がタイプ τ の項ならば $\lambda x.f$ はタイプ $\tau\to\sigma$ の項である。

  5) f が $\tau\to\sigma$ の項で、x がタイプ τ の項ならば、$f(x)$ はタイプ σ の項である。

  6) g がタイプ τ の項で、h がタイプ $\mathrm{o}\to(\tau\to\tau)$ の項ならば、$\rho[g,h]$ はタイプ $\mathrm{o}\to\tau$ の項である。

  7) 自由変数を持たない項を、原始帰納的汎関数と呼ぶ。
- TT の論理式 (*wff* または *formula*) は、同じタイプの項 f と g について、$f=g$ の形をしたものを素論理式 (*atomic formula*) として、これに論理記号 $\neg, \wedge, \vee, \supset$ を何回か施して得られるものである。

- TT の公理と推論規則

  直観主義自然数論 (HA) の公理と推論規則を認める。但し限量記号 $(\forall, \exists)$ に関する推論規則を除く。

  以下の公理を追加する。

  - λ に関して
    $(\lambda x.f(x))(t) = f(t)$
    $\lambda x.f(x) = f$
  - f と g とが束縛変数以外は同型の時、$f = g$
  - ρ に関して
    $\rho[g,h](0) = g$
    $\rho[g,h](s(n)) = h(n, \rho[g,h](n))$
    ここに n はタイプ o である。
  - 関数の合成に関して
    $(f_1 = f_2) \wedge (g_1 = g_2) \supset f_1(g_1) = f_2(g_2)$

(定義終)

定義 6 QT (Quantified Terms)

- QT の言語: TT の言語に論理記号 $\forall$ と $\exists$ を追加する。
- QT の論理式: TT の論理式を次の様に拡張して得られるもの。すなわち、A が既に定義された論理式、x が変数ならば $\forall x A$ と $\exists x A$ はともに QT の論理式である。
- QT の公理と推論規則は規定せず。

QT において公理と推論規則を規定しない理由は、QT は論理式の変換の中間段階として利用されるだけで、証明図の変換は HA から TT へ直接行なうからである。図 3.4参照。以上でゲーデル解釈のための準備が整った。

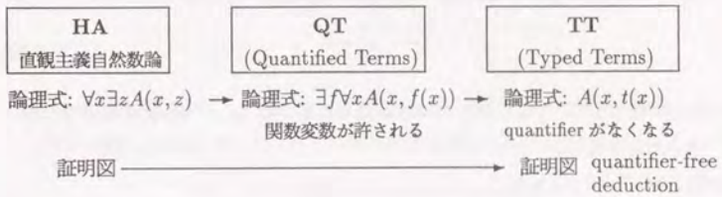

図 3.4: HA, QT, TT の関係 (論理式と証明図)

定義 7 ゲーデル解釈 (HA の論理式を QT の論理式で解釈する)

- 項 (term) のゲーデル解釈

  HA における項は、QT におけるタイプ o の項になる。(厳密に言えば HA における原始帰納的関数を表す記号を QT における原始帰納的汎関数で置き換える。)

- 論理式のゲーデル解釈

  1. 論理式 $\mathcal{A}$ が限量記号 (quantifier)$\forall$, $\exists$ を含まない時には、$\mathcal{A}$ の中の各素論理式 $u = v$ に含まれている項 u と v とを各々ゲーデル解釈して等号で結んだものが $\mathcal{A}$ のゲーデル解釈である。
  2. 論理式 $\mathcal{A}$ が限量記号を含む時には、次の様に帰納的に定義する。$\mathcal{A}$ の一部である $\mathcal{B}, \mathcal{C}$ のゲーデル解釈が既に次の様に定義されていると仮定する。(注: $\exists x$, $\forall y$, $\exists u$, $\forall v$ の一部が実際には無い場合もある。)
     $\mathcal{B}$ の解釈が $\exists x \forall y B(x,y)$、$\mathcal{C}$ の解釈が $\exists u \forall v C(u,v)$ とする。
     (i) $\mathcal{A}$ が $\neg\mathcal{B}$ であれば、$\exists z \forall x \neg B(x, z(x))$.
     (ii) $\mathcal{A}$ が $\mathcal{B} \wedge \mathcal{C}$ であれば、$\exists x \exists u \forall y \forall v (B(x,y) \wedge C(u,v))$.
     (iii) $\mathcal{A}$ が $\mathcal{B} \vee \mathcal{C}$ であれば、$\exists d \exists x \exists u \forall y \forall v \{(d = 0 \wedge B(x,y)) \vee (d \neq 0 \wedge C(u,v))\}$.
     ここに新しい変数 d は $\mathcal{B}$ 側が成り立つか $\mathcal{C}$ 側が成り立つかを示す。
     (iv) $\mathcal{A}$ が $\mathcal{B} \supset \mathcal{C}$ であれば、$\exists w \exists z \forall x \forall v \{B(x, z(x,v)) \supset C(w(x), v)\}$.
     また $\mathcal{F}(a)$ のゲーデル解釈が既に定義されていて、$\exists x \forall y F(a,x,y)$ である時、
     (v) $\mathcal{A}$ が $\forall u \mathcal{F}(u)$ であれば、$\exists z \forall u \forall y F(u, z(u), y)$.

(vi) $\mathcal{A}$ が $\exists u\mathcal{F}(u)$ であれば、$\exists u\exists x\forall y F(u,x,y)$.

(定義終)

上記の定義の中では $\mathcal{B} \supset \mathcal{C}$ の解釈の部分が最も複雑である。この解釈は次の様に段階的に考えると分かり易い。

1. $\exists x\forall y B(x,y) \supset \exists u\exists v C(u,v)$
2. $\forall x\{\forall y B(x,y) \supset \exists u\exists v C(u,v)\}$
3. $\forall x\exists u\{\forall y B(x,y) \supset \forall v C(u,v)\}$
4. $\forall x\exists u\forall v\{\forall y B(x,y) \supset C(u,v)\}$
5. $\forall x\exists u\forall v\exists y\{B(x,y) \supset C(u,v)\}$
6. $\exists w\forall x\forall v\exists y\{B(x,y) \supset C(w(x),v)\}$
7. $\exists w\exists z\forall x\forall v\{B(x,z(x,v)) \supset C(w(x),v)\}$

上のゲーデル解釈の定義によって、HA の論理式が QT の論理式に変換されることが分かる。ところで、QT の論理式に変換されたものは必ず

$$\exists x\forall y A(x,y)$$

という形をしている ($\exists\forall$-formula)。これをさらに適当なタイプの項 t と適当なタイプの自由変数 f とを用いて、

$$A(t,f)$$

と書き換えると、これは TT の論理式となる。厳密に言うと、$\exists x\forall y A(x,y)$ には自由変数が含まれていたかも知れない (そのタイプは必ず o である)。したがって自由変数 f は元から含まれていた自由変数とは異なるものを選ばなくてはならない。また項 t は元来含まれていた自由変数を含んで良い。

以上により、最終的に TT の論理式が得られる。そして次の重要な定理が成り立つ。

定理 4 (Gödel) 論理式 A が、HA において証明可能ならば、A を解釈して得られた TT の論理式は TT において証明可能である。

Proof この定理を証明するためには、HA の公理と推論規則を変換したものが TT で証明可能であることを示せば良い。ここでは証明の細部には立ち入らずに、プログラム合成の立場から見て興味深いものをピックアップする。

i) HA の推論規則が次の形をしている時、

$$\frac{\mathcal{A} \quad \mathcal{A} \supset \mathcal{B}}{\mathcal{B}}$$

$\mathcal{A}$ の解釈が $\exists x\forall y A(x,y)$ であり、$\mathcal{B}$ の解釈が $\exists u\forall v B(u,v)$ であると仮定する。推論規則の解釈の結果は次のような TT における推論規則に該当する。

$$\frac{A(t_1,y_1) \quad A(x_2,t_2(x_2,v_2)) \supset B(t_3(x_2),v_2)}{B(t_3(t_1),v_2)}$$

この TT の推論規則は二つのプログラム t_1 と t_3 を合成した関数 $t_3(t_1)$ に対応している。

ii) HA の推論規則が次の形をしている時、解釈後の TT の推論規則は上と同様であるがもう少し複雑になる。(省略)

$$\frac{\mathcal{A} \supset \mathcal{B} \quad \mathcal{B} \supset \mathcal{C}}{\mathcal{A} \supset \mathcal{C}}$$

iii) HA の公理が $(\mathcal{A} \vee \mathcal{A}) \supset \mathcal{A}$ の形の時、$\mathcal{A}$ の解釈が $\exists x\forall y A(x,y)$ とすると解釈後の TT の論理式は次の様になる。

$$\{(d=0 \wedge A(x_1,y_3)) \vee (d \neq 0 \wedge A(x_2,y_3))\} \supset A(\begin{bmatrix} x_1 \text{ if } d=0 \\ x_2 \text{ if } d \neq 0 \end{bmatrix}, y_3)$$

これはプログラムの条件式 (あるいは条件文) に対応する。

iv) HA の公理が $\mathcal{A} \supset (\mathcal{A} \wedge \mathcal{A})$ の時、上と同様である。

v) HA の推論規則が数学的帰納法の時、

$$\frac{\mathcal{A}(0) \quad \forall x(\mathcal{A}(x) \supset \mathcal{A}(s(x)))}{\forall x\mathcal{A}(x)}$$

$\mathcal{A}$ の解釈が $\exists z\forall w A(z,w,x)$ とすると、TT では次の推論規則になる。

$$\frac{A(t_0,w,0) \quad A(z_1,t_1(x,z_1,w_2),x) \supset A(t_2(x,z_1),w_2,s(x))}{A(\rho[t_0,t_2](x),w,x)}$$

ここに $\rho[t_0, t_2]$ は次の様に定義される。

$$\begin{cases} \rho[t_0, t_2](0) &= t_0 \\ \rho[t_0, t_2](s(n)) &= t_2(n, \rho[t_0, t_2](n)) \end{cases}$$

これはプログラムの原始帰納法に対応する。

(proof 終)

なお HA の公理と推論規則は、限量記号に関するものを除けば、そのまま TT の公理あるいは推論規則にもなっている (定義 5参照)。しかし HA の公理と推論規則とを変換したものが、TT において「元の自分の姿」と同じになるとは限らない。したがって、限量記号を含まない公理・推論規則に関してもゲーデルの定理は自明ではないので証明が必要である。ゲーデルの定理の証明の全容は Shütte[50], 竹内・八杉 [52], Troelstra[57] 等に載っている。

次節ではゲーデルの定理を応用してプログラムの合成を行なう方法について述べる。

3.3 ゲーデル解釈を応用したプログラム合成

前節で述べたゲーデルの定理は一般の論理式に対して成り立つ。本節では定理を利用してプログラムの合成を行うために、仕様を表す論理式に対してゲーデルの解釈を適用する。プログラムの合成は以下の手順に従う (この記述は Goto[65] に基づく)。

1. 求めるプログラムの仕様を表す論理式を作る。例えば割算のプログラムであれば次の論理式が仕様を表す。$\forall x_1 \forall x_2 \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}$, ここに x_1 は被除数、x_2 は除数、z_1 が商で z_2 が剰余である。

2. 直観主義自然数論 HA の体系の中で (1) の論理式を証明する。
   上の割算の論理式の証明は細部を省略すると次の様な形をしている。

$$\frac{\forall x_2 QR(0, x_2) \qquad \forall x_2 QR(x, x_2) \supset \forall x_2 QR(s(x), x_2)}{\forall x_1 \forall x_2 QR(x_1, x_2)} \text{帰納法}$$

   ここに $QR(a, b)$ は $QR(a, b) \equiv \exists z_1 \exists z_2 (a = b \cdot z_1 + z_2 \wedge z_2 < b)$ という論理式の略記である。(QR は Quotient and Remainder の意味)

3. 仕様を表す論理式を有限のタイプの体系である TT に変換する。この TT の論理式の証明を次に構成する。

4. HA の証明図を TT に変換する。TT の証明図の中にはプログラムに相当する情報が汎関数の形で含まれている。

5. 上の汎関数を抽出すれば所望のプログラムが求まる。

6. 最後に、有限のタイプの汎関数を Lisp の関数として表す。このステップが実際のプログラムの形を決定する。我々が実際に使用した Lisp は LIPQ[54] である。汎関数を Lisp で表す方法は次の通り。

   1) 有限のタイプの汎関数としての自然数 0 は、Lisp の数値アトム 0 で表す。

   2) タイプ σ の自由変数は Lisp の変数で表す。

   3) f がタイプ o の項で、Lisp では F で表されているとする。この時 $s(f)$(f の successor) は Lisp の `(ADD1 F)`, `(+ F)` 等で表す。ここに `(+ F)` は LIPQ 固有の関数表記である。

4) f がタイプ σ の項であり、Lisp では F で表されているものとする。さらに x がタイプ τ の項であり、Lisp では X で表されているものとする。この時 $\lambda x.f$ はタイプ $\tau \to \sigma$ の項であるが、Lisp では (LAMBDA (X) F) で表現される。

5) f がタイプ $\tau \to \sigma$ の項で Lisp では F で表されているとする。また x はタイプ τ の項で Lisp では X で表されているものとする。この時、$f(x)$ はタイプ σ の項であり、Lisp での表現は (F X) となる。

6) g がタイプ τ の項であり、Lisp では G で表されるとする。また h はタイプ $\mathrm{o} \to (\tau \to \tau)$ の項であり、Lisp では H で表されるものとする。この時、$\rho[g,h]$ はタイプ $\mathrm{o} \to \tau$ の項であり、Lisp では次のように表される。

```
(LABEL R001
   (LAMBDA (X)
      (COND ((ZEROP X) G)
            (T (H (SUB1 X) (R001 (SUB1 X)))))))
```

または

```
(DE R001 (X)
    (COND ((ZEROP X) G)
          (T (H (SUB1 X) (R001 (SUB1 X))))))
```

7) 自由変数を全く含まない項、すなわち原始帰納的汎関数は Lisp のプログラムとして表される。

このような手順にしたがって実際にプログラムの合成を行なった例を次節に示す。なお付録 A.2にも合成の例を報告しておいた。

3.4 プログラム合成の例

本節では次の順序で例題を説明する。

1. 仕様を表す論理式
2. 論理式の HA における証明図
3. TT において証明すべき論理式 (1 の論理式の TT における解釈)
4. TT における証明図 (2 の証明図の TT における解釈)
5. プログラムに対応する汎関数
6. Lisp によるプログラムの表現と動作例

例題:

$\boxed{1}$ 仕様を表す論理式:

$\forall x_1 \forall x_2 \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}$

この論理式は今までに繰返し説明した割算の仕様を表す。

$\boxed{2}$ HA における証明図: 紙面の都合で帰納法の basis と step とに分割して書く。

$$\dfrac{\begin{array}{c}\text{この部分証明}\\ \text{を basis と呼ぶ} \Rightarrow \vdots \\ \mathcal{A}(\dot{0})\end{array} \qquad \begin{array}{c}\text{この部分証明}\\ \vdots \Leftarrow \text{を step と呼ぶ} \\ \forall x(\mathcal{A}(x) \supset \mathcal{A}(s(x)))\end{array}}{\forall x_1 \mathcal{A}(x_1)}$$

basis の詳細

$$\dfrac{\dfrac{\dfrac{\overset{(\text{公理})}{0 = x_2 \cdot 0 + 0} \qquad \overset{(\text{公理})}{x_2 \neq 0 \supset 0 < x_2}}{x_2 \neq 0 \supset (0 = x_2 \cdot 0 + 0 \wedge 0 < x_2)}\,(*1)}{x_2 \neq 0 \supset \exists z_1 \exists z_2 (0 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}\,(*2)}{\forall x_2 \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (\mathbf{0} = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}}$$

ここに証明図の最下段の論理式の中で太字のゼロ (**0**) は帰納法の対象となる項を表す。なお basis の証明図の中で $0 = x_2 \cdot 0 + 0$ および $x_2 \neq 0 \supset 0 < x_2$ は本来の公理 (図 3.1 ~ 3.3)

には含まれないが、簡単に導くことができるためにここでは公理として扱っている。なお (*1), (*2) 等の推論規則も本来の推論規則ではなく、後に示す派生規則である。

step の詳細

$$\dfrac{\overset{(\text{公理})}{z_2 < x_2 \supset s(z_2) < x_2 \vee s(z_2) = x_2}}{\begin{array}{c}(x_2 \neq 0 \wedge x = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \supset \\ (x_2 \neq 0 \wedge x = x_2 \cdot z_1 + z_2 \wedge s(z_2) < x_2) \vee (x_2 \neq 0 \wedge x = x_2 \cdot z_1 + z_2 \wedge s(z_2) = x_1)\end{array}} \ (*3)$$

$$\begin{array}{cc} \cup \text{ 中略} & \cup \text{ 中略} \\ (s(x) = x_2 \cdot z_1 + s(z_2) \wedge s(z_2) < x_2) & (s(x) = x_2 \cdot s(z_1) + 0 \wedge 0 < x_2) \\ \cup \text{ 中略} & \cup \text{ 中略} \\ \exists z_1 \exists z_2 (s(x) = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) & \exists z_1 \exists z_2 (s(x) = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\exists z_1 \exists z_2 (s(x) = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \quad \exists z_1 \exists z_2 (s(x) = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}{(x_2 \neq 0 \wedge x = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \supset \exists z_1 \exists z_2 (s(x) = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}\ (*4)}{(x = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \supset \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (s(x) = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}}}{\exists z_1 \exists z_2 (x = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \supset \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (s(x) = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}}}{\begin{array}{c}\forall x_2 \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (x = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\} \supset \\ \forall x_2 \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (s(x) = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}\end{array}}\ (*5)}{\begin{array}{c}\forall \mathbf{x} \{\forall x_2 \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (\mathbf{x} = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\} \supset \\ \forall x_2 \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (\mathbf{s(x)} = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}\}\end{array}}$$

上の証明図の最下段の論理式の中で太字の $\mathbf{x}$ および $\mathbf{s(x)}$ は帰納法を適用する対象の項を表す。なお step の証明図においても $z_2 < x_2 \supset s(z_2) < x_2 \vee s(z_2) = x_2$ を公理と見なした他、(*4) と (*5) の派生規則を用いた。下に示す派生規則は本来の推論規則 (図 3.1, 図 3.2) を用いて簡単に導くことができる。

$$(*1)\ \dfrac{A \quad B \supset C}{B \supset (A \wedge C)} \qquad (*2)\ \dfrac{A \supset B(t)}{A \supset \exists x B(x)} \qquad (*3)\ \dfrac{A \supset B \vee C}{(D \wedge A) \supset (D \wedge B) \vee (D \wedge C)}$$

$$(*4)\ \dfrac{A \supset (B \vee C) \quad B \supset D \quad C \supset D}{A \supset D} \qquad (*5)\ \dfrac{A \supset (B \supset C)}{(B \supset A) \supset (B \supset C)}$$

3 TT において証明すべき論理式:(1 の論理式の TT における解釈)

$x_2 \neq 0 \supset (x_1 = x_2 \cdot T_1(x_1, x_2) + T_2(x_1, x_2) \wedge T_2(x_1, x_2) < x_2)$

T_1, T_2 は TT における証明図が出来上がれば具体的に定まる。

4 TT における証明図: (2 の証明図の TT における解釈) TT における帰納法の推論は下の形をしている。$\mathcal{A}(x) \equiv \exists z \forall w A(z, w, x)$ と略記する。

$$\dfrac{\begin{array}{c}\text{basis} \Rightarrow \vdots \\ \mathcal{A}(t_0, w, 0)\end{array} \quad \begin{array}{c}\vdots \Leftarrow \text{step} \\ \mathcal{A}(z_1, t_1(x, z_1, w_2), x) \supset \mathcal{A}(t_2(x, z_1), w_2, s(x))\end{array}}{A(\rho[t_0, t_2](x), w, x)}$$

basis の詳細

$$\dfrac{0 = x_2 \cdot 0 + 0 \qquad x_2 \neq 0 \supset 0 < x_2}{x_2 \neq 0 \supset (0 = x_2 \cdot 0 + 0 \wedge 0 < x_2)}$$

step の詳細

$$\dfrac{z_2 < x_2 \supset s(z_2) < x_2 \vee s(z_2) = x_2}{\begin{array}{c}(x_2 \neq 0 \wedge x = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \supset \\ (x_2 \neq 0 \wedge x = x_2 \cdot z_1 + z_2 \wedge s(z_2) < x_2) \vee (x_2 \neq 0 \wedge x = x_2 \cdot z_1 + z_2 \wedge s(z_2) = x_2)\end{array}}$$

$$\begin{array}{cc} \cup \text{ 中略} & \cup \text{ 中略} \\ (s(x) = x_2 \cdot z_1 + s(z_2) \wedge s(z_2) < x_2) & (s(x) = x_2 \cdot s(z_1) + 0 \wedge 0 < x_2) \end{array}$$

$$\dfrac{\begin{array}{c}(x_2 \neq 0 \wedge x = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \supset \\ (s(x) = x_2 \cdot \begin{bmatrix} z_1 & \text{if } s(z_2) < x_2 \\ s(z_1) & \text{if } s(z_2) = x_2 \end{bmatrix} + \begin{bmatrix} s(z_2) & \text{if } s(z_2) < x_2 \\ 0 & \text{if } s(z_2) = x_2 \end{bmatrix} \wedge \begin{bmatrix} s(z_2) \\ 0 \end{bmatrix} < x_2)\end{array}}{\dfrac{\begin{array}{c}(x = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \supset \\ \{x_2 \neq 0 \supset (s(x) = x_2 \cdot \begin{bmatrix} z_1 \\ s(z_1) \end{bmatrix} + \begin{bmatrix} s(z_2) \\ 0 \end{bmatrix} \wedge \begin{bmatrix} s(z_2) \\ 0 \end{bmatrix} < x_2)\}\end{array}}{\begin{array}{c}\{x_2 \neq 0 \supset (\mathbf{x} = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\} \supset \\ \{x_2 \neq 0 \supset (\mathbf{s(x)} = x_2 \cdot \begin{bmatrix} z_1 \\ s(z_1) \end{bmatrix} + \begin{bmatrix} s(z_2) \\ 0 \end{bmatrix} \wedge \begin{bmatrix} s(z_2) \\ 0 \end{bmatrix} < x_2)\}\end{array}}}$$

5 プログラムに対応する汎関数: 3 の T_1 と T_2 とは 4 を参照すると次の様に定まる。

$$\begin{cases} T_1(0, x_2) = 0 \\ T_1(s(n), x_2) = \begin{cases} T_1(n, x_2) & \text{if } s(T_2(n, x_2)) < x_2 \\ s(T_1(n, x_2)) & \text{if } s(T_2(n, x_2)) = x_2 \end{cases} \end{cases}$$

$$\begin{cases} T_2(0, x_2) = 0 \\ T_2(s(n), x_2) = \begin{cases} s(T_2(n, x_2)) & \text{if } s(T_2(n, x_2)) < x_2 \\ 0 & \text{if } s(T_2(n, x_2)) = x_2 \end{cases} \end{cases}$$

6 Lisp によるプログラムの表現と動作例:

```
(DE T1 (X1 X2)
 (COND ((ZEROP X1) 0)
       ((LESSP (+ (T2 (- X1) X2)) X2) (T1 (- X1) X2))
       ((EQ (+ (T2 (- X1) X2)) X2) (+ (T1 (- X1) X2))) ))

(DE T2 (X1 X2)
 (COND ((ZEROP X1) 0)
       ((LESSP (+ (T2 (- X1) X2)) X2) (+ (T2 (- X1) X2)))
       ((EQ (+ (T2 (- X1) X2)) X2) 0) ))
```

下のプログラムの動作例においてはDIV1という名前の関数を作って割算の商と剰余を同時に表示する。(このDIV1という関数は合成されたものではない。)

```
(DE DIV1 (X1 X2) (LIST (T1 X1 X2) (T2 X1 X2)))

*(DIV1 5 3)
(1 2)

*(DIV1 6 2)
(3 0)

*(DIV1 0 5)
(0 0)

*(DIV1 0 0)
(0 0)
```

なお上記の例題においては幾つかの仮定を公理と見なしたため、厳密に言うと前節までの理論と異なる点が一箇所ある。これを扱うためには精密な議論が必要とされるので、6.2節で改めて取り上げる。

本節ではゲーデルの解釈に基づくプログラムの合成法を説明するために基本的な例題を説明した。より進んだプログラムの合成の例は付録 A.2に報告してある。

第 4 章

無限リストの論理的な解釈

論理式をプログラムのように見立てて直接に実行する方法について考察する。この方法では計算すべきアルゴリズムを論理式で記述する。その論理式を証明し、証明図を Prawitz のいわゆる**正規化** (normalization) 技法を用いて実行する。この考え方は、論理的な証明系をあたかもプログラミング言語のように見なすことになる。その意味では証明図の正規化の技法は Prolog に代表される論理プログラミングの考え方に近い。ただし正規化と論理プログラミングとは細部において異なっている。

4.1 正規化による計算機構

証明図の正規化という技法は数理論理学の分野では古くから知られている。例えば、Prawitz は 1965 年に著した「自然演繹法」という本 [43] の中で様々な論理体系に対して正規化を適用して見せている。正規化の手法が情報工学的に見て重要であるのは次の事実による。

> 『もし $\exists z A(z)$ という形の論理式が構成的論理の中で証明可能であるとすると、その証明図に対して正規化の処理を施すことによって、z に対する具体的な解 t を見つけることができる。t は $A(t)$ を満たす項である。』

これは、本論文の序論の中で構成的論理の特徴として述べた「具体的に見つける方法」が正規化によって実現できることを示している。このような計算機構は正に論理プログラミングと呼ぶのがふさわしい。

証明図の正規化は自然演繹法の定式化を用いると奇麗に説明できる。既に第 2章で自然演繹法の推論規則を説明した。また自然演繹法に基づく自然数論の公理と数学的帰納法の

推論規則も第2章で説明済みである。本章でも図2.3、図2.7の公理と推論規則を用いて証明図を組み立てる。

以下で取り扱う典型的な証明図は図4.1の様な形をしている。この証明図の中では数学的帰納法 IND が使われている。図4.1の結論、すなわち証明された論理式は $\forall x \exists z(x+y=z)$ である。この論理式は自然数における加算を表現している。

$$
\dfrac{\dfrac{\dfrac{0+y=y}{\exists z(0+y=z)}\exists I \quad \dfrac{[\exists z(a+y=z)] \quad \dfrac{\dfrac{[a+y=c] \quad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{a+y=c \supset s(a)+y=s(c)}\forall E}{s(a)+y=s(c)}\supset E}{\exists z(s(a)+y=z)}\exists I}{\exists z(s(a)+y=z)}\exists E}{\exists z(b+y=z)}\mathrm{IND}}{\forall x \exists z(x+y=z)}\forall I
$$

図 4.1: 数学的帰納法を用いた証明図

図4.1の証明図には二つの仮定 (assumption) が使われている。一つは左上の $0+y=y$ であり、二番目は右上の $\forall xwz(x+w=z \supset s(x)+w=s(z))$ である。この他にカギカッコ [ ] で囲まれた論理式が二つ存在する。$[a+y=c]$ と $[\exists z(a+y=z)]$ である。これらは仮定とは見なされない。なぜなら下の意味で「落とされた」仮定であるからである。より正確に言うと、$[a+y=c]$ は推論規則 $\exists E$ によって落とされ、$[\exists z(a+y=z)]$ は推論規則 IND によって落とされる。

定義 8 推論規則 $\vee E$, $\supset I$, $\exists E$, IND の適用においてカギカッコで囲まれた仮定は推論規則によって**落とされた** (*discharged*) 仮定と呼ばれる。証明図の結論は落とされていない仮定 (*open assumption* という) に依存するが、落とされた仮定には依存しない。

なお各推論規則の前提 (横棒の上にある論理式) の種類を区別しておいた方が良い。

定義 9 論理記号を除去する推論規則 (E-rule) の前提の内で、除去される推論規則 ($\wedge$, $\vee$, $\supset$, $\forall$, $\exists$) を含む論理式の方をその推論規則の**主前提** (*major premise*) と呼ぶ。主前提以外の前提を**副前提** (*minor premise*) と呼ぶ。論理記号を導入する推論規則 (I-rule) および人の推論規則および自然数論固有の基本規則の前提はすべて主前提と呼ぶ。数学的帰納法 (IND) の推論規則の二つの前提は、ともに便宜上副前提と見なす。

さて証明図の正規化とは、一口に言えば自然演繹法の証明図の中の冗長な部分 (redundant part) を消去するものである (Prawitz[43], Troelstra[57])。冗長部を除去するには次の様なリダクション規則を適用する。なお序論では $\supset$ に関するリダクション規則を述べたが本章では $\supset$ のリダクションは使用しない。

リダクション規則

R1. $\forall$ リダクション規則: 証明図の中で $\forall$I-rule と $\forall$E-rule とが連続して適用されている部分があれば、その二つの規則の適用を両方とも消去することにより証明図が簡略化される。下の証明図の中で $\Pi(a)$ は a をパラメータとする部分証明図を表す。

$$
\dfrac{\dfrac{\begin{array}{c}\Pi(a)\\ A(a)\end{array}}{\forall x(A(x))}\forall I}{A(t)}\forall E \quad \text{is reduced to} \quad \begin{array}{c}\Pi(t)\\ A(t)\end{array}
$$

$$\Rightarrow$$

R2. $\exists$ リダクション規則: 証明図の中で $\exists$I-rule と $\exists$E-rule とが連続して使われている部分があれば、その二つの規則の適用を両方とも消去することにより証明図を簡略化できる。下の証明において Π と $\Pi'(a)$ はともに部分証明図である。$\Pi'(a)$ の中の a はパラメータである。

$$
\dfrac{\dfrac{\begin{array}{c}\Pi\\ A(t)\end{array}}{\exists x(A(x))}\exists I \quad \begin{array}{c}[A(a)]\\ \Pi'(a)\\ C\end{array}}{C}\exists E \quad \text{is reduced to} \quad \begin{array}{c}\Pi\\ [A(t)]\\ \Pi'(t)\\ C\end{array}
$$

$$\Rightarrow$$

R3. IND リダクション規則: IND 推論規則のリダクションは帰納法の対象となる項 t が 0 である場合、あるいは t が s(t) の形である場合に証明図を簡略化する。

$$
\dfrac{\begin{array}{c}\Pi_0\\ A(0)\end{array} \quad \begin{array}{c}[A(a)]\\ \Pi(a)\\ A(s(a))\end{array}}{A(0)}\mathrm{IND} \quad \Rightarrow \quad \begin{array}{c}\Pi_0\\ A(0)\end{array}
$$

および

$$
\dfrac{\begin{array}{c}\Pi_0\\ A(0)\end{array} \quad \begin{array}{c}[A(a)]\\ \Pi(a)\\ A(s(a))\end{array}}{A(s(t))}\mathrm{IND} \quad \Rightarrow \quad \dfrac{\begin{array}{c}\Pi_0\\ A(0)\end{array} \quad \begin{array}{c}[A(a)]\\ \Pi(a)\\ A(s(a))\end{array}}{\begin{array}{c}[A(t)]\\ \Pi(t)\\ A(s(t))\end{array}}\mathrm{IND}
$$

R4. ⅄リダクション規則: ⅄の規則に対するリダクション規則は推論規則の結論となる論理式の複雑度 (complexity) を小さくするように作用する。ここに論理式の複雑度とは、論理式に含まれる論理記号の数のことである。この規則を反復して適用すれば、⅄規則の結論は最終的に素論理式 (atomic formula) になる。ここに素論理式とは $\wedge, \vee, \supset, \forall, \exists$ などの論理記号を全く含まない論理式のことである。下には A∧B の場合のみを示す。他の論理記号に対しても同様に働く。

$$\dfrac{\dfrac{\Pi}{\text{⅄}}}{\text{A}\wedge\text{B}}\,\text{⅄} \quad\Rightarrow\quad \dfrac{\dfrac{\dfrac{\Pi}{\text{⅄}}}{\text{A}}\,\text{⅄} \qquad \dfrac{\dfrac{\Pi}{\text{⅄}}}{\text{B}}\,\text{⅄}}{\text{A}\wedge\text{B}}$$

上の⅄のリダクションは本章の例題では使われないが、5.2節の定理7の説明に関係があるためここで述べておいた。

定義 10 それ以上リダクション規則が適用できない証明図を正規形 (*normal form*) であると言う。

以下には正規形である証明図の性質を述べるが、その前に Harrop 論理式と呼ばれる論理式のクラスを定義する。Harrop という名前は人名に由来する。

定義 11 **Harrop 論理式**の集合は以下のように帰納的に定義される。

1. 素論理式は *Harrop* 論理式である。
2. A と B が既に *Harrop* 論理式であると分かっている時、$A \wedge B$ と $\forall x A(x)$ とは *Harrop* 論理式である。
3. B が既に *Harrop* 論理式であると分かっている時、$A \supset B$ は *Harrop* 論理式である。ここに A は任意の論理式で良い。

結局 Harrop 論理式とは $\vee$ と $\exists$ の論理記号を含まない論理式のことである。ただし $A \supset B$ の左辺 A には含まれていても良い。$\vee$ と $\exists$ とを特別扱いする理由は、ちょうど第2章で述べた構成的論理および直観主義論理の特徴に呼応する。すなわち Harrop 論理式とは計算上の情報を含まない論理式のことである。論理記号 $\vee$ と $\exists$ には「具体的な方法」が伴っていなければならないことに注意しよう。

次の定理が証明図の正規化の重要な性質を述べている。

定理 5 $\exists z A(z)$ を結論とする証明図 Π が次の条件を満たす時、

1. Π は正規形である。
2. Π の仮定がすべて *Harrop* 論理式である。

その証明図の結論を導く最後の推論規則 (一番下の推論規則) は必ず $\exists$I であり、$A(t)$ から $\exists z A(z)$ を導く規則になっている。

Proof Troelstra[57] 参照。

定理に述べられている正規形の性質を用いると図 4.1の証明図を利用して二つの整数の和を計算することができる。図 4.1の仮定はいずれも Harrop 論理式であることに注意しよう。証明図の正規形を求めるためのリダクションの様子を下に示す。例題中のステップ 1 では ∀x に s(0) が代入される。これが一方の入力になる。s(0) は数字の 1 を表す。例題中の証明図の中で $\boxed{\text{box}}$ に囲まれた推論規則がリダクション規則で消去される対象である。ステップ 5 で証明図が正規形になるが、この時の最後の推論規則は ∃I である。このように証明図の正規化を応用して ∃z(s(0)+y=z) の解を導くことができる。具体的な解は s(y) であり、確かに s(0)+y=s(y) を満たす。

例題: 正規化による加算の計算

ステップ 1: 図 4.1の証明図の下に ∀E 規則を追加する。そして ∀x の x に s(0) を代入する。$\boxed{\forall\text{I}}$ と $\boxed{\forall\text{E}}$ が ∀ リダクションの対象となる。

$$\dfrac{\dfrac{\dfrac{0+y=y}{\exists z(0+y=z)}\,\exists\text{I} \qquad \dfrac{[\exists z(a+y=z)] \qquad \dfrac{\dfrac{[a+y=c] \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{a+y=c \supset s(a)+y=s(c)}\,\forall\text{E}}{s(a)+y=s(c)}\,\supset\text{E}}{\exists z(s(a)+y=z)}\,\exists\text{I}}{\exists z(s(a)+y=z)}\,\exists\text{E}}{\exists z(b+y=z)}\,\text{IND}}{\dfrac{\forall x\exists z(x+y=z)}{\exists z(s(0)+y=z)}\,\boxed{\forall\text{E}}}\,\boxed{\forall\text{I}}$$

ステップ 2: 次に IND リダクションを適用する。

$$\dfrac{\dfrac{0+y=y}{\exists z(0+y=z)}\,\exists I \qquad \dfrac{[\exists z(a+y=z)] \qquad \dfrac{\dfrac{[a+y=c] \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{a+y=c \supset s(a)+y=s(c)}\,\forall E}{s(a)+y=s(c)}\,\supset E}{\exists z(s(a)+y=z)}\,\exists I}{\exists z(s(a)+y=z)}\,\exists E}{\exists z(s(0)+y=z)}\,\boxed{\mathrm{IND}}$$

ステップ 3: IND リダクションの結果、証明図のサイズは大きくなった。しかし帰納法の対象となる項は s(0) から 0 に簡約化された。

$$\dfrac{\dfrac{\dfrac{0+y=y}{\exists z(0+y=z)}\,\exists I \qquad \dfrac{[\exists z(a+y=z)] \qquad \dfrac{\dfrac{[a+y=c] \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{a+y=c \supset s(a)+y=s(c)}\,\forall E}{s(a)+y=s(c)}\,\supset E}{\exists z(s(a)+y=z)}\,\exists I}{\exists z(s(a)+y=z)}\,\exists E}{\exists z(0+y=z)}\,\boxed{\mathrm{IND}} \qquad \dfrac{\dfrac{[0+y=c] \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{0+y=c \supset s(0)+y=s(c)}\,\forall E}{s(0)+y=s(c)}\,\supset E}{\exists z(s(0)+y=z)}\,\exists I}{\exists z(s(0)+y=z)}\,\exists E$$

ステップ 4: 再び IND リダクションを適用する。今度は帰納法の対象となる項が 0 であるから basis のみが取出される。

$$\dfrac{\dfrac{0+y=y}{\exists z(0+y=z)}\,\boxed{\exists I} \qquad \dfrac{\dfrac{[0+y=c] \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{0+y=c \supset s(0)+y=s(c)}\,\forall E}{s(0)+y=s(c)}\,\supset E}{\exists z(s(0)+y=z)}\,\exists I}{\exists z(s(0)+y=z)}\,\boxed{\exists E}$$

ステップ 5: 最後に ∃ リダクションを適用すると正規形となる。証明図の最後の推論規則は ∃I である。

$$\dfrac{\dfrac{0+y=y \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{0+y=y \supset s(0)+y=s(y)}\,\forall E}{s(0)+y=s(y)}\,\supset E}{\exists z(s(0)+y=z)}\,\exists I$$

(例題終)

上の例題のステップ 2 で帰納法の対象となる項は s(0)=1 である。一般に帰納法の対象となる項が n の場合にリダクションの系列は下の様になる。本論文では n が ω となる場合を取扱うが、その準備として次節では推論規則を改良する。

$$\dfrac{A(0) \quad \Pi(a)}{A(n)}\ \mathrm{IND} \Rightarrow \dfrac{A(0) \quad \Pi(a)}{\begin{array}{c}A(n\dot{-}1)\\ \Pi(n\dot{-}1)\\ A(n)\end{array}} \Rightarrow \ \cdots \ \Rightarrow \begin{array}{c}A(0)\\ \Pi(0)\\ A(1)\\ \vdots\\ A(n\dot{-}1)\\ \Pi(n\dot{-}1)\\ A(n)\end{array}$$

4.2 新しい推論規則

証明図の正規化による計算は理論的に見れば完璧なものである。しかし以下に見るように応用上の観点からは改善の余地が残されている。本節では正規化による計算の有用性を高めるために、新たな観点から改良を加える。我々の方法は自然演繹法の推論規則(第2章の図2.3)の中の∃除去の規則(∃E)をGentzenの原型からBorkowskiとSłupeckiの規則と呼ばれる規則に変えることである。(以下BS規則と略す)BS規則はPrawitzの教科書[43]の末尾の解説の中でGentzenの定式化に対するバリエーションの一つとして説明されている。[43]のような論理学の教科書の立場では主として論理式の証明可能性を問題とするから、BS規則は単なる記法上のバリエーションに過ぎない。しかし本節で明らかにするように、証明図の正規化を行なう場合にはBS規則と伝統的なGentzenの規則との間には本質的な差が生じる。すなわちBS規則を用いることにより、正規化の処理を一歩進めることができる。なお以下では説明の便宜上、Gentzenの規則からBS規則へ置き換える操作をあたかもリダクション規則の一つであるかのように取り扱う(Goto[74], [85])。

定義 12 BorkowskiとSłupeckiの∃除去(elimination)規則

$$\frac{\exists x(\mathrm{A}(x,y))}{\mathrm{A}(\alpha_y,y)}\ \mathrm{BS}$$

Borkowski と Słupecki の規則では新しい対象記号として部分パラメータ(原論文では *ambiguous name*)という要素を使用する。部分パラメータ(例えば上の α)は対象変数を添字として持つ場合がある。上の場合には y が添字である。

部分パラメータというものはSkolem関数に似ている。事実本研究の初期の段階では、部分パラメータに相当する働きをSkolem関数で説明していた(Goto[73])。しかし、その後の検討で部分パラメータを用いた説明の方が論理的により適切な説明を与えることが判明した。なお付録A.3では部分パラメータをタイプ理論の立場で正当化している。推論規則の形が変更されたのに伴って、リダクション規則もBS規則をカバーするように拡張される。下のR5は単に∃E規則をBS規則で置き換えるもの(上述)である。またR2′のリダクション規則は以前のR2リダクション規則を置き換えるものである。

R5. BSリダクション規則: Gentzenの∃除去の規則をBS規則で置き換える。

$$\dfrac{\exists x(A(x)) \quad \begin{array}{c}[A(a)]\\ \vdots\\ C\end{array}}{C}\ \exists E \qquad \text{is reduced to} \quad \Rightarrow \qquad \dfrac{\exists x(A(x))}{\begin{array}{c}A(\alpha)\\ \vdots\\ C\end{array}}\ BS$$

R2′. BS-∃I リダクション規則: 証明図の中で∃I規則の直後にBS規則が続いている場合には、その二つの推論規則の適用を省略することにより証明図を簡略化できる。

$$\dfrac{\dfrac{A(t)}{\exists x(A(x))}\ \exists I}{A(\alpha)}\ BS \qquad \text{is reduced to} \quad \Rightarrow \qquad A(t)$$

厳密に言うと論理式 $A(\alpha)$ は単一の論理式とは限らない。付録A.3にR2′規則の詳細な説明を記す。

具体的な証明の中でBS規則がどのように使われるかを観察してみる。以前の例題と同様に図4.1を用いて加算を実行する例を、今度はBS規則を用いて正規化する。次の例題の正規化の過程、すなわちリダクションのステップを前節の例題と比較すると興味深い結果が得られる。

例題: BS規則による正規化

ステップ 1: 図4.1の証明図に対して今度はBS規則を先に適用する。

$$\dfrac{\dfrac{\dfrac{\dfrac{0+y=y}{\exists z(0+y=z)}\ \exists I \quad \dfrac{[\exists z(a+y=z)] \quad \dfrac{\dfrac{[a+y=c] \quad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{a+y=c \supset s(a)+y=s(c)}\ \forall E}{s(a)+y=s(c)}\ \supset E}{\exists z(s(a)+y=z)}\ \exists I}{\exists z(s(a)+y=z)}\ \boxed{\exists E}}{\exists z(b+y=z)}\ IND}{\forall x\exists z(x+y=z)}\ \forall I}{\exists z(s(0)+y=z)}\ \forall E$$

ステップ 2: $\boxed{\forall I}$ と $\boxed{\forall E}$ とが∀リダクションにより省かれる。

$$
\dfrac{\dfrac{\dfrac{0+y=y}{\exists z(0+y=z)}\exists I \qquad \dfrac{\dfrac{\dfrac{[\exists z(a+y=z)]}{a+y=\alpha_{a,y}}\ \mathrm{BS} \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{a+y=\alpha_{a,y} \supset s(a)+y=s(\alpha_{a,y})}\forall E}{s(a)+y=s(\alpha_{a,y})}\supset E}{\exists z(s(a)+y=z)}\exists I}{\exists z(b+y=z)}\ \mathrm{IND}}{\dfrac{\forall x\exists z(x+y=z)}{\exists z(s(0)+y=z)}\boxed{\forall E}}\boxed{\forall I}
$$

ステップ 3: IND リダクションが適用される。

$$
\dfrac{\dfrac{0+y=y}{\exists z(0+y=z)}\exists I \qquad \dfrac{\dfrac{\dfrac{[\exists z(a+y=z)]}{a+y=\alpha_{a,y}}\ \mathrm{BS} \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{a+y=\alpha_{a,y} \supset s(a)+y=s(\alpha_{a,y})}\forall E}{s(a)+y=s(\alpha_{a,y})}\supset E}{\exists z(s(a)+y=z)}\exists I}{\exists z(s(0)+y=z)}\boxed{\mathrm{IND}}
$$

ステップ 4: まだ正規形にはなっていないが、部分解が現れる。

$$
\dfrac{\dfrac{\dfrac{0+y=y}{\exists z(0+y=z)}\exists I \qquad \dfrac{\dfrac{\dfrac{[\exists z(a+y=z)]}{a+y=\alpha_{a,y}}\ \mathrm{BS} \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{a+y=\alpha_{a,y} \supset s(a)+y=s(\alpha_{a,y})}\forall E}{s(a)+y=s(\alpha_{a,y})}\supset E}{\exists z(s(a)+y=z)}\exists I}{\exists z(s(0)+y=z)}\boxed{\mathrm{IND}} \qquad \dfrac{\dfrac{\exists z(0+y=z)}{0+y=\alpha_{0,y}}\ \mathrm{BS} \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{0+y=\alpha_{0,y} \supset s(0)+y=s(\alpha_{0,y})}\forall E}{s(0)+y=s(\alpha_{0,y})}\supset E}{\exists z(s(0)+y=z)}\exists I
$$

ステップ 5: ∃I-BS リダクション規則が適用される。

$$
\dfrac{\dfrac{\dfrac{0+y=y}{\exists z(0+y=z)}\boxed{\exists I} \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{0+y=\alpha_{0,y} \supset s(0)+y=s(\alpha_{0,y})}\forall E}{s(0)+y=s(\alpha_{0,y})}\boxed{\mathrm{BS}}\supset E}{\exists z(s(0)+y=z)}\exists I
$$

ステップ 6: 前節の例題のステップ 5 と全く同じ証明図が得られる。

$$
\dfrac{\dfrac{0+y=y \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{0+y=y \supset s(0)+y=s(y)}\forall E}{s(0)+y=s(y)}\supset E}{\exists z(s(0)+y=z)}\exists I
$$

(例題終)

今度の例題では、最初に Gentzen の推論規則が BS 規則によって置き換えられる (R5 による)。この時に部分パラメータとして $\alpha_{a,y}$ が導入される。後にステップ 4 で $\alpha_{a,y}$ の一部は $\alpha_{0,y}$ に置き換えられる。これは IND の推論規則を適用する際に a に 0 が代入されるからである。∃I-BS リダクション規則 (R2′) はステップ 5 で適用される。その後のリダクションの進行は前節の例題と同様である。表 4.1 と表 4.2 には前節の例題と本節の BS 規則を用いたリダクションのまとめを掲げてある。明らかに二つの表の最後の解の欄は等しい。

ステップ	1	2	3	4	5
正規形?	No	No	No	No	Yes
最後の推論規則	∀E	IND	∃E	∃E	∃I
解	–	–	–	–	s(y)

表 4.1: 前節の例題のリダクション

ここで注目すべき点は、表 4.2 のステップ 4, 5, 6 の証明図の中で各々最後の推論規則が ∃I になっていることである。したがってステップ 6 に至る以前に ∃z に対する解 (項) を取り出すことが可能である。

ステップ	1	2	3	4	5	6
正規形?	No	No	No	No	No	Yes
最後の推論規則	∃E	∃E	IND	∃I	∃I	∃I
解	–	–	–	$s(\alpha_{0,y})$	$s(\alpha_{0,y})$	s(y)

表 4.2: BS 規則を用いたリダクション

ただしステップ 4, 5 での解は部分パラメータ $\alpha_{0,y}$ を含む項 $s(\alpha_{0,y})$ である。この $\alpha_{0,y}$ は後に y に置き換えられる。このような部分パラメータを含む解を部分解と呼ぶ。それは

解の部分的な情報を与えているからである。$s(\alpha_{0,y})$ の場合には項の先頭が s であること、すなわち解の形が $s(\cdot)$ であるという情報を与えている。

この部分解の現象を論理的に説明する目的で、我々は**分離された証明図**という概念を導入する。つまり正規形の証明図の性質を利用するために、証明図の方を幾つかの部分証明に分解しておき、正規形の定理を個別に適用するのである。

定義 13　証明図 Π の結論となっている論理式を $Con(\Pi)$ で表わす。

i) 証明図 Π の部分証明図 Π_0 が**分離可能**であるとは、Π_0 のいかなる仮定も $Con(\Pi_0)$ の下では落とされない (*not discharged*) 場合を言う。

ii) 証明図 Π の分離可能な部分証明図 Π_0 をその結論の論理式 $Con(\Pi_0)$ で置き換えて得られる証明図を**分離された証明図**と言い、$\Pi - \Pi_0$ と書く。$\Pi - \Pi_0$ を $Con(\Pi_0)$ **において**分離された証明図と呼ぶ。

分離された証明図が正しく自然演繹法の証明図になっていることは自明とは言えず、証明を必要とする。

定理 6　分離された証明図 $\Pi - \Pi_0$ は自然演繹法の証明になっている。論理式 $Con(\Pi_0)$ は自然数論の固有の公理、または $\Pi - \Pi_0$ の落ちていない (*open*) 仮定となる。

Proof 証明を付録 A.4に掲げる。

例えば本節の例題のリダクションのステップ 4 の証明図は $0{+}y{=}\alpha_{0,y}$ のところで分離できる。分離された証明は正規形となっており、その最後の推論規則は ∃I である。

$$\dfrac{\dfrac{0+y=\alpha_{0,y} \qquad \dfrac{\forall xwz(x+w=z \supset s(x)+w=s(z))}{0+y=\alpha_{0,y} \supset s(0)+y=s(\alpha_{0,y})}\,\forall E}{s(0)+y=s(\alpha_{0,y})}\,\supset E}{\exists z(s(0)+y=z)}\,\exists I$$

定義 14　証明 Π が**疑似正規形**であるとは、Π の部分証明 $\Pi_0, \Pi_1, \ldots, \Pi_n$ が存在して分離された証明 $(((\Pi - \Pi_0) - \Pi_1) \ldots - \Pi_n)$ が正規形になることを言う。

本節の例題ではステップ 4 および 5 の証明図が疑似正規形になる。さらに $0{+}y{=}\alpha_{0,y}$ という論理式は Harrop 論理式である。したがってステップ 4 および 5 に定理 5を適用することができる。これが $0{+}y{=}\alpha_{0,y}$ で分離された証明図に対して解 $s(\alpha_{0,y})$ が得られる理由である。

これに対して前節の例題の方では、ステップ 4 の証明図が疑似正規形になっている。この証明図では $\exists z(0{+}y{=}z)$ で分離された証明図が正規形になる。しかし、定理 5はこの分離された証明図には適用できない。なぜなら $\exists z(0{+}y{=}z)$ は Harrop 論理式ではないからである。

このような簡単な加算の例題の比較を通して、BS 規則が∃記号を除去することにより non-Harrop 論理式を Harrop 論理式に変換する働きがあることが分かった。この性質は証明図の正規化を考える上で極めて重要である。

4.3 無限の要素を含む正規化

上述の新しい証明図の正規化手法は、証明図の中に無限大の要素が含まれる時に威力を発揮する。我々は自然数論を拡大して無限大の整数を考え、さらに無限長の長さを持つリストについて論じる。

無限の概念を表すのに最も良く使われている概念は**順序数** (ordinal number) である。順序数は確かに無限の概念の一側面を良く表わしている。しかし、すぐ後に述べる理由により、リストの長さを表すのに順序数を用いるのは不適切であることが分かる。

順序数は普通の自然数の拡張として考えることができる。普通の自然数を次の様に表記できることは良く知られている。ここに ϕ は空集合を表わす。

$$
\begin{aligned}
0 &= \phi \\
1 &= 0 \cup \{0\} = \{0\} \\
2 &= 1 \cup \{1\} = \{0,1\} \\
3 &= 2 \cup \{2\} = \{0,1,2\} \\
\vdots &= \vdots \\
n+1 &= n \cup \{n\} = \{0,1,2,\ldots,n\} \\
\vdots &= \vdots
\end{aligned}
$$

この表記法では自然数 n は 0 から $n-1$ までの集合と同一視される。同様の考え方を反復すると、自然数全体の集合 $\{0,1,2,\ldots,n,\ldots\}$ が得られるが、これが ω に相当する。ω は普通の自然数ではなく、自然数を拡張した**順序数**である。なお普通の自然数は順序数の一種でもある。順序数を用いてリストの長さを表現すれば、次の様になる。ここに () は空リストを表す。

リスト	長さ
()	0
(a_0)	1
$(a_0\ a_1)$	2
$(a_0\ a_1\ a_2)$	3
...	...
$(a_0\ a_1\ a_2\ \ldots)$	ω

ここで重要なことは、リストの長さは伸び縮みすることである。例えば $cons(x, (a_0\ a_1\ a_2\ \ldots))$, x は任意の要素, の結果のリストの長さは $\omega+1$ となる。この場合には問題が生じない。

しかし長さが縮む場合を考えると、次のような問題が起こる。例えば $cdr((a_0\ a_1\ a_2\ \ldots))$ の長さは ω よりも 1 だけ短い。これを $\omega-1$ と表すことができるかというと、残念ながら $\omega-1=\omega$ となってしまう。つまり ω より 1 だけ小さい順序数と言うものは存在しないのである。

そこで我々は $\omega-1$ が意味を持つような無限の概念を検討した。結論を言うと超準解析 (Robinson[46]) で使用される ω を応用すれば $\omega-1$ が意味を持ち、リストの長さを表すのに適していることが分かった。超準解析における ω の考え方を簡単に説明しよう。以下では自然数の無限列を考える。ここに無限とは自然数の濃度のことを指す。具体的に言うと普通の自然数 0 は次のように 0 が反復して現れる無限列と同一視される。

$$0 = \langle 0,0,0,\ldots,0,\ldots\rangle$$

同様に各々の自然数は次のような定数列と同一視される。

$$
\begin{aligned}
1 &= \langle 1,1,1,\ldots,1,\ldots\rangle \\
2 &= \langle 2,2,2,\ldots,2,\ldots\rangle \\
\vdots &= \vdots \\
n &= \langle n,n,n,\ldots,n,\ldots\rangle \\
\vdots &= \vdots
\end{aligned}
$$

このような枠組みの中で ω は次の上昇列と同一視される。この列の i 番目の要素は i である。

$$\omega = \langle 0,1,2,\ldots,n,\ldots\rangle$$

ここで肝心な点は、超準解析のモデルにおいて無限列を比較する場合には、有限個の要素が違っていても良いことである。例えば次の列は最初の要素が 1 となっているので上の ω とは多少異なるが、有限個の要素を除いて等しいので同じ ω を表すものと見なす。

$$\omega = \langle 1,1,2,\ldots,n,\ldots\rangle$$

さて、ω の重要な性質として無限の概念を表わしていることを確かめる必要がある。例えば $\omega > 100$ という比較を考える。その結果は、列の最初の 0 番目から 99 番目までは 100 の方が大きいから答えは F(*false*) である。100 番目の要素は互いに等しいから F である。101 番目以降の要素については ω の要素の方が大きいから T(*true*) となる。

要素	0	1	2	...	99	100	101	...	n	...
ω	0	1	2	...	99	100	101	...	n	...
100	100	100	100	...	100	100	100	...	100	...
比較	F	F	F	...	F	F	T	...	T	...

ここで再び有限個の要素を無視して考える。すると上の比較の結果は、有限個 (0 から 100 番目まで) を除いては T (*true*) になる。したがって $\omega > 100$ は成り立つ。同様にして ω はいかなる有限の自然数よりも大きいことが分かる。

以上はモデル論的な説明であった。以下の本論では定理の証明図を論じるのであるから、公理論的に ω を導入する。そのためには無限大を意味する新しい定数 ω を自然数の体系に追加する。さらに ω に関する固有の公理として $\{0 < \omega,\ 1 < \omega,\ 2 < \omega, \ldots\}$ を追加する。この公理は無限大の整数が存在することを主張している。このように拡大された自然数論の世界でも数学的帰納法は有効に働く。(なお本論文の 6.3節では我々と独立に研究を進めていた Martin-Löf による定式化 [33], [34] と我々の方法を比較する。)

無限大の整数を含む例題の中では、次に示す下降型帰納法 (going-down induction) が使われる。これに対して通常の帰納法を上昇型帰納法と呼ぶことがある。

下降型の数学的帰納法

$$\frac{A(t) \qquad \begin{array}{c}[A(s(a))]\\ \vdots \\ A(a)\end{array}}{A(u)}\ \mathrm{IND}\downarrow \qquad \text{ここに } t \geq u \text{ とする}$$

下降型帰納法とコンピュータ・プログラムとの関係は Manna と Waldinger の論文 [29] の中でも述べられていた。彼らの論文は、下降型帰納法を用いて合成されるプログラムの形について論じている。本章で示すのは、下降型帰納法が証明図の正規化においても興味深い応用を持つことである。まず最初に下降型帰納法を普通の帰納法から導く。その証明図が後に正規化の対象となる。

下降型の帰納法自体は論理学の教科書にも説明が載っていることが多い。ただし殆どの論理学の教科書は古典論理に基づいているから、普通の (上昇型) 帰納法から下降型帰納法を導く証明も古典論理の体系の中で行なわれている。その場合の証明は著しく簡単である。しかし我々の論理体系は直観主義自然数論であり、その中で証明しなければ正規化の対象とはならない。直観主義自然数論の中で普通の上昇型帰納法から下降型帰納法を導く証明は難しくはないが多少長い。図 4.2、図 4.3参照。図 4.2の中では普通の帰納法

IND と EQ という推論規則を用いている。EQ は自然数論に固有の推論規則である (第2章図 2.7参照)。関数記号として "$\dot{-}$" が使われているが、これは自然数の範囲の減算、すなわち 0 よりは小さくならない減算である。$x \dot{-} 0 = x,\ x \dot{-} s(y) = pd(x \dot{-} y)$, ここに pd とは predecessor function すなわち $pd(0) = 0,\ pd(s(x)) = x$ となる関数である。

$$\dfrac{\dfrac{t \geq u \qquad t \geq u \supset (t \dot{-} (t \dot{-} u) = u)}{t \dot{-} (t \dot{-} u) = u}\ \supset\mathrm{E} \qquad \dfrac{A(t) \qquad \dfrac{\forall x(x = x \dot{-} 0)}{t = t \dot{-} 0}\ \forall\mathrm{E}}{A(t \dot{-} 0)}\ \mathrm{EQ} \qquad \begin{array}{c}[A(t \dot{-} a)]\\ \mathtt{Subproof(t,a)}\\ A(t \dot{-} s(a))\end{array}}{A(t \dot{-} (t \dot{-} u))}\ \mathrm{IND}}{A(u)}\ \mathrm{EQ}$$

図 4.2: 下降型帰納法の証明の主要部分

図 4.2における帰納法の対象となる項は $t \dot{-} (t \dot{-} u)$ の中の $t \dot{-} u$ である。なお Subproof(t,a) と書いてある部分証明の詳細は図 4.3に証明してある。

$$\dfrac{\dfrac{\forall x(x = 0 \vee x \neq 0)}{t \dot{-} a = 0 \vee t \dot{-} a \neq 0}\ \forall\mathrm{E} \qquad \dfrac{[A(t \dot{-} a)] \qquad \dfrac{\dfrac{[t \dot{-} a = 0] \quad \dfrac{\dfrac{\forall xy(x \dot{-} y = 0 \supset x \dot{-} y = x \dot{-} s(y))}{t \dot{-} a = 0 \supset t \dot{-} a = t \dot{-} s(a)}\ \forall\mathrm{E}}{t \dot{-} a = t \dot{-} s(a)}\ \supset\mathrm{E}}{A(t \dot{-} s(a))}\ \mathrm{EQ} \qquad \dfrac{[A(t \dot{-} a)] \qquad \dfrac{\dfrac{[t \dot{-} a \neq 0] \quad \dfrac{\dfrac{\forall xy(x \dot{-} y \neq 0 \supset x \dot{-} y = s(x \dot{-} s(y)))}{t \dot{-} a \neq 0 \supset t \dot{-} a = s(t \dot{-} s(a))}\ \forall\mathrm{E}}{t \dot{-} a = s(t \dot{-} s(a))}\ \supset\mathrm{E}}{A(s(t \dot{-} s(a)))}\ \mathrm{EQ}}{\begin{array}{c}\Pi(t \dot{-} s(a))\\ A(t \dot{-} s(a))\end{array}}}{A(t \dot{-} s(a))}}{A(t \dot{-} s(a))}\ \vee\mathrm{E}$$

図 4.3: 部分証明 Subproof(t,a) の詳細

次に証明図のリダクションを考える。図 4.2に対しては従来の IND リダクションが有効に働く。すなわち、もし t=u の場合には図 4.2で言えば $t \dot{-} u = 0$ に該当するから従来のリダクション規則 R3 により帰納法の step の部分が枝刈りされて証明図が簡略化される。簡略化された証明図の結論は A(u) と書いてあるが、t=u と仮定しているから A(t) に等しい。また $A(t \dot{-} 0)$ とも等しい。なぜなら $t \dot{-} 0 = t$ が成り立つからである。

$$\dfrac{\begin{array}{c}\vdots\\ t \dot{-} 0 = u\end{array} \qquad \dfrac{\dfrac{A(t) \qquad \dfrac{\forall x(x = x \dot{-} 0)}{t = t \dot{-} 0}\ \forall\mathrm{E}}{A(t \dot{-} 0)}\ \mathrm{EQ} \qquad \begin{array}{c}[A(t \dot{-} a)]\\ \mathtt{Subproof(t,a)}\\ A(t \dot{-} s(a))\end{array}}{A(t \dot{-} 0)}\ \mathrm{IND}}{A(u)}\ \mathrm{EQ} \quad \Rightarrow \quad \dfrac{t \dot{-} 0 = u \qquad \dfrac{A(t) \qquad \dfrac{\forall x(x = x \dot{-} 0)}{t = t \dot{-} 0}\ \forall\mathrm{E}}{A(t \dot{-} 0)}\ \mathrm{EQ}}{A(u)}\ \mathrm{EQ}$$

このように簡略化された証明図 (“⇒” の右辺) は正規形になっている。ここで A(u) と A(t) および A(t∸0) を同一視すれば右側の証明図全体は単一の論理式 A(t) に等しい。このように考えると次の IND↓ リダクション規則を得る。

R6a. IND↓ リダクション規則

$$\dfrac{\begin{array}{c}\Pi_0\\ A(t)\end{array}\quad\begin{array}{c}[A(s(a))]\\ \Pi(a)\\ A(a)\end{array}}{A(t)}\ \mathrm{IND}\downarrow \quad\Rightarrow\quad \begin{array}{c}\Pi_0\\ A(t)\end{array}$$

また t>u の場合には t∸u>0 となるから t∸u=s(t∸s(u)) と書き直せることを利用して図 4.2の証明図に従来の IND リダクション規則を適用することができる。すなわち帰納法の対象となる項が s(t) となる場合の規則が有効である。

$$\dfrac{\begin{array}{c}\vdots\\ t\dot{-}s(t\dot{-}s(u))=u\end{array}\quad \dfrac{\dfrac{A(t)\quad \dfrac{\forall x(x=x\dot{-}0)}{t=t\dot{-}0}\,\forall E}{A(t\dot{-}0)}\,\mathrm{EQ}\quad \begin{array}{c}[A(t\dot{-}a)]\\ \mathtt{Subproof}(t,a)\\ A(t\dot{-}s(a))\end{array}}{A(t\dot{-}s(t\dot{-}s(u)))}\,\mathrm{IND}}{A(u)}\,\mathrm{EQ}$$

is reduced to
⇒

$$\dfrac{\begin{array}{c}\vdots\\ t\dot{-}s(t\dot{-}s(u))=u\end{array}\quad \dfrac{\dfrac{A(t)\quad \dfrac{\forall x(x=x\dot{-}0)}{t=t\dot{-}0}\,\forall E}{A(t\dot{-}0)}\,\mathrm{EQ}\quad \begin{array}{c}[A(t\dot{-}a)]\\ \mathtt{Subproof}(t,a)\\ A(t\dot{-}s(a))\end{array}}{\begin{array}{c}A(t\dot{-}(t\dot{-}s(u)))\\ \mathtt{Subproof}(t,t\dot{-}s(u))\\ A(t\dot{-}s(t\dot{-}s(u)))\end{array}}\,\mathrm{IND}}{A(u)}\,\mathrm{EQ}$$

リダクション後の証明図の中の項 t∸(t∸s(u)) は t∸(t∸s(u))=s(u) のように計算される。ここでは t≥s(u) ならば s(u)=t∸(t∸s(u)) という事実を使った。このリダクション後の証明図を IND↓ を用いて整理すると下のようなリダクション規則にまとめることができる。

R6b. IND↓ リダクション規則

$$\dfrac{\begin{array}{c}\Pi_0\\ A(t)\end{array}\quad\begin{array}{c}[A(s(a))]\\ \Pi(a)\\ A(a)\end{array}}{A(u)}\ \mathrm{IND}\downarrow \ \Rightarrow\ \dfrac{\begin{array}{c}\Pi_0\\ A(t)\end{array}\quad\begin{array}{c}[A(s(a))]\\ \Pi(a)\\ A(a)\end{array}}{\begin{array}{c}[A(s(u))]\\ \Pi(u)\\ A(u)\end{array}}\ \mathrm{IND}\downarrow$$

この新しいリダクション規則 R6a と R6b を用いて下の図 4.4の証明図を正規化することができる。項 [x|w] は $cons(x,w)$ の Prolog 風の表現である。したがって [a|b] は $cons(a,b)$ の意味である。図 4.4の証明図の意味は無限のリスト (ストリーム) [0|[1|[2|[3| ...]]]] が存在することを主張している。この証明図には二つの仮定があり、図 4.4の右上方の仮定は整数 s(x) に対して w というリストが存在するならば、整数 x に対してリスト [x|w] が存在するということを主張している。また左方の仮定は整数 ω に対して空リスト [ ] が存在する、というものである。ここに ω は無限大の整数である。ω =s(s(s(...(s(0))...)))

$$\dfrac{\mathrm{integers}(\mathrm{s}(\omega),[\,])\quad \dfrac{[\exists z(\mathrm{integers}(\mathrm{s}(a),z))]\quad \dfrac{\dfrac{[\mathrm{integers}(\mathrm{s}(a),b)]\quad \dfrac{\forall xw(\mathrm{integers}(\mathrm{s}(x),w)\supset \mathrm{integers}(x,[x|w]))}{\mathrm{integers}(\mathrm{s}(a),b)\supset \mathrm{integers}(a,[a|b])}\,\forall E}{\mathrm{integers}(a,[a|b])}\,\supset E}{\exists z(\mathrm{integers}(a,z))}\,\exists I}{\exists z(\mathrm{integers}(a,z))}\,\exists E}{\exists z(\mathrm{integers}(0,z))}\,\mathrm{IND}\downarrow$$

図 4.4: ストリームの存在証明

図 4.4に対する正規化の処理を概観すると次のようになる。最初に BS リダクション規則が適用されて変数 b が部分パラメータ α_a で置き換えられる。次に IND↓ リダクション規則が適用されて a に 0 が代入されるから α_a は α_0 となる。この項が最初の部分解を構成する。すなわち $[0|\alpha_0]$ が部分解である。リダクションのステップは同様に進行して、次のような部分解の列を生成する。

$$[0|\alpha_0],\ \ [0|[\mathrm{s}(0)|\alpha_{s(0)}]],\ \ [0|[\mathrm{s}(0)|[\mathrm{s}(\mathrm{s}(0))|\alpha_{s(s(0))}]]],\ldots$$

最終的な解は無限長の長さを持つ。これはリダクションのステップが無限に続くからである。下の図では $A(n)\equiv\exists z(\mathrm{integers}(n,z))$ と略記してある。

$$\dfrac{A(\omega)\quad \Pi(a)}{A(0)}\ \mathrm{IND}\downarrow \ \Rightarrow\ \dfrac{A(\omega)\quad \Pi(a)}{\begin{array}{c}A(s(0))\\ \Pi(0)\\ A(0)\end{array}}\ \Rightarrow\ \ldots\ \Rightarrow\ \begin{array}{c}A(\omega)\\ \Pi(\omega\dot{-}1)\\ A(\omega\dot{-}1)\\ \vdots\\ A(s(0))\\ \Pi(0)\\ A(0)\end{array}$$

最終的な解は無限のリストすなわちストリームというデータ構造である。この解は下の Concurrent Prolog[49] のプログラムの一行目が生成するストリームと同じものである。なお図 4.5のプログラムは竹内 [53] の中の例題を一部書き直して引用した。

```
(1)  integers(N,[N|S]) :- N1:=N+1 | integers(N1,S).
(2)  even([A|L],[A|W]) :- 0 is A mod 2 | even(L?,W).
(3)  even([A|L],W) :- otherwise | even(L?,W).
(4)  :- integers(0,S),even(S?,W).
```

図 4.5: Concurrent Prolog のプログラム

事実、図 4.4の証明図の最初の仮定は図 4.5の一行目のステートメントと論理的に同じものである。次節で図 4.5のプログラム全体と同じ動作を正規化による計算で実行する。

最後に次の点に注意しておこう。図 4.4の証明図を正規化する際に BS 規則を用いなくても正規化のステップを進めることはできる。ただし、部分解を求める訳にはいかない。BS 規則を用いないで解を得るためには無限ステップの間待たなければならない。この例題は簡単なものではあるけれども、BS 規則と部分解の有効性を示している。

4.4 並列論理プログラミングとの比較

Concurrent Prolog のプログラムを正規化による計算で実行する。図 4.5のステートメント (1) は既に図 4.4の証明図で取り扱われていた。ステートメント (2) と (3) とは下の図 4.6の中の二つの仮定 **Ass(2)** と **Ass(3)** に論理的に同値である。なお図中の even(l,c) は「l は自然数のリストであり、c は l の中の偶数よりなるリストである」を表わす。ここでは even([], []) を公理として認めている。図 4.6の中の部分証明 II は結論 0=(a mod 2) ∨ 0≠(a mod 2) を証明している。II の中では数学的帰納法 IND が使われている筈であるが、ここでは詳細な説明を省く。

Ass(2): (0=(a mod 2)∧even(l,c)) ⊃ even([a|l],[a|c])
Ass(3): (0≠(a mod 2)∧even(l,c)) ⊃ even([a|l],c)

$$
\dfrac{\dfrac{\text{even([],[])}}{\exists w(\text{even([],w))}} \quad \dfrac{[\exists w(\text{even(l,w)})] \quad \dfrac{\dfrac{\text{II}}{0=(\text{a mod 2})\vee 0\neq(\text{a mod 2})} \quad \dfrac{\dfrac{\dfrac{[0=(\text{a mod 2})]\ [\text{even(l,c)}]}{0=(\text{a mod 2})\wedge\text{even(l,c)}}\ \textbf{Ass(2)}}{\text{even([a|l],[a|c])}}}{\exists w(\text{even([a|l],w))}} \quad \dfrac{\dfrac{\dfrac{[0\neq(\text{a mod 2})]\ [\text{even(l,c)}]}{0\neq(\text{a mod 2})\wedge\text{even(l,c)}}\ \textbf{Ass(3)}}{\text{even([a|l],c)}}}{\exists w(\text{even([a|l],w))}}}{\exists w(\text{even([a|l],w))}}\ \vee E}{\exists w(\text{even([a|l],w))}}\ \exists E}{\exists w(\text{even(t,w))}}\ \text{IND[]}
$$

図 4.6: 偶数列に関する証明

この証明図 4.6の中ではリストに関する帰納法 (IND[] と表示した) を用いている。リストに関する帰納法は、普通の帰納法 IND の拡張版であり、次のように自然数とリストの構成要素とを対応付けて得られる。$0 \leftrightarrow []$、右辺は空リストの意味、および $\mathrm{s}(y) \leftrightarrow [x|y]$、右辺は $cons(x, y)$ の意味。IND[] に関するリダクション規則も同様に導かれるが、詳細は省略する。帰納法の対象となる項 $[x|y]$ が s(t) のように扱われる。

$$
\dfrac{\text{A([])} \quad \begin{array}{c}[\text{A(l)}]\\ \vdots \\ \text{A([a|l])}\end{array}}{\text{A(t)}}\ \text{IND[]}
$$

最終的に二つの証明図 4.4と 4.6とを結合した証明図をつくる。これが Concurrent Prolog のステートメント (4) に対応する。次に示す例題は、結合した証明図のリダクション

のステップの最初の5ステップ目までを表示したものである。この部分だけを観察すれば、以下の計算の様子も伺い知ることができる。IND↓を使った部分証明図(図の左側)がリスト(ストリーム)を生成し、右側のIND[]を用いた部分証明の側がリストを受け取って処理をする。二つの部分証明のリダクションは原則として独立に行なうことができる。

例題: Concurrent Prolog と同等の計算

ステップ 1: 結合した証明図。BS 規則で $\boxed{\exists E}$ 規則を置換える。

$$\dfrac{\dfrac{\mathrm{integers}(\omega,[\,]) \qquad \dfrac{[\exists z(\mathrm{integers}(s(a),z)]}{\exists z(\mathrm{integers}(a,z))}}{\exists z(\mathrm{integers}(0,z))}\,\mathrm{IND}\downarrow \qquad \dfrac{[\mathrm{integers}(0,b)] \qquad \dfrac{\exists w(\mathrm{even}([\,],w)) \quad \dfrac{[\exists w(\mathrm{even}(l,w))]}{\exists w(\mathrm{even}([a|l],w))}}{\exists w(\mathrm{even}(b,w))}\,\mathrm{IND}[\,]}{\mathrm{integers}(0,b)\wedge\exists w(\mathrm{even}(b,w))}\,\wedge\mathrm{I}}{\exists z(\mathrm{integers}(0,z)\wedge\ \exists w(\mathrm{even}(z,w)))}\,\exists\mathrm{I}}{\exists z(\mathrm{integers}(0,z)\wedge\exists w(\mathrm{even}(z,w)))}\,\boxed{\exists E}$$

ステップ 2: BS 規則により部分パラメータ β が導入される。$\boxed{\mathrm{IND}\downarrow}$ が次にリダクションの対象となる。

$$\dfrac{\dfrac{\dfrac{\mathrm{integers}(\omega,[\,]) \qquad \dfrac{[\exists z(\mathrm{integers}(s(a),z)]}{\exists z(\mathrm{integers}(a,z))}}{\exists z(\mathrm{integers}(0,z))}\,\boxed{\mathrm{IND}\downarrow}}{\mathrm{integers}(0,\beta)}\,\mathrm{BS} \qquad \dfrac{\exists w(\mathrm{even}([\,],w)) \quad \dfrac{[\exists w(\mathrm{even}(l,w))]}{\exists w(\mathrm{even}([a|l],w))}}{\exists w(\mathrm{even}(\beta,w))}\,\mathrm{IND}[\,]}{\mathrm{integers}(0,\beta)\wedge\exists w(\mathrm{even}(\beta,w))}\,\wedge\mathrm{I}}{\exists z(\mathrm{integers}(0,z)\wedge\exists w(\mathrm{even}(z,w)))}\,\exists\mathrm{I}$$

ステップ 3: IND↓ リダクションの後には図4.4で説明した部分パラメータ α_0 が現れる。

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathrm{integers}(0,[0|\alpha_0])}\,\boxed{\exists I}}{\exists z(\mathrm{integers}(0,z))}\,\boxed{BS}}{\mathrm{integers}(0,\beta)} \qquad \dfrac{\exists w(\mathrm{even}([\,],w)) \quad \dfrac{[\exists w(\mathrm{even}(l,w))]}{\exists w(\mathrm{even}([a|l],w))}}{\exists w(\mathrm{even}(\beta,w))}\,\mathrm{IND}[\,]}{\mathrm{integers}(0,\beta)\wedge\exists w(\mathrm{even}(\beta,w))}\,\wedge\mathrm{I}}{\exists z(\mathrm{integers}(0,z)\wedge\exists w(\mathrm{even}(z,w)))}\,\exists\mathrm{I}$$

ステップ 4: ∃I-BS リダクションにより β に $[0|\alpha_0]$ が代入される。

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathrm{integers}(0,[0|\alpha_0])} \qquad \dfrac{\exists w(\mathrm{even}([\,],w)) \quad \dfrac{[\exists w(\mathrm{even}(l,w))]}{\exists w(\mathrm{even}([a|l],w))}}{\exists w(\mathrm{even}([0|\alpha_0],w))}\,\boxed{\mathrm{IND}[\,]}}{\mathrm{integers}(0,[0|\alpha_0])\wedge\exists w(\mathrm{even}([0|\alpha_0],w))}\,\wedge\mathrm{I}}{\exists z(\mathrm{integers}(0,z)\wedge\exists w(\mathrm{even}(z,w)))}\,\exists\mathrm{I}$$

ステップ 5: 証明図の右側(図4.6に相当する部分)の $\exists w(\mathrm{even}(l,w))$ に対応して部分パラメータ γ_l が導入される。w に対する部分解の形は一旦 $[0|\gamma_{\alpha_0}]$ の形となる。証明図の左側で生成されたリスト $[0|\alpha_0]$ の先頭の要素0は偶数(even number)であるから解のリスト $[0|\gamma_{\alpha_0}]$ の中に現れる。

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathrm{integers}(0,[0|\alpha_0])} \qquad \dfrac{\mathrm{even}([0|\dot{\alpha}_0],[0|\gamma_{\alpha_0}])}{\exists w(\mathrm{even}([0|\alpha_0],w))}\,\exists\mathrm{I}}{\mathrm{integers}(0,[0|\alpha_0])\wedge\exists w(\mathrm{even}([0|\alpha_0],w))}\,\wedge\mathrm{I}}{\exists z(\mathrm{integers}(0,z)\wedge\exists w(\mathrm{even}(z,w)))}\,\exists\mathrm{I}$$

(例題終)

ここで興味深いのは図4.5の Concurrent Prolog のプログラムには read-only annotation という記法が使われていることである。例えば "L?" とか "S?" のように "?" を伴っている変数がその記法に該当する。この記法の働きは一種の同期機構を実現することである。これに対して我々の正規化による計算では特殊な記法を使わなくて済む。これは正規化の計算では、リストを受け取る側(IND[])のリダクションを行う以前にリストを生成する側(IND↓)からの部分解を受け取らなくてはならないからである。換言すれば、正規化の計算ではオーバーラン(データが揃わない内に計算が始まること)は生じない。これは BS 規則によって部分解を作る場合にも成立する。この点が正規化による計算の特徴である。強調して言えば正規化の計算は純論理的である。これに比べると Concurrent Prolog の read-only annotation は論理的な機構とは言えない。

第 5 章

無限リストを応用したプログラムの解析法

本章ではプログラムのトレースを表現するための新しい方法を提案する。この方法を用いるとトレースをあたかも証明図のように表現できる。この性質を用いて二つのプログラムが同値であることをトレースの上で判定できる。

一般にトレースの情報量はプログラムの情報量よりも少ない。しかし無限長のリストに対するトレースを想定すれば、プログラムに相当する情報をトレースで表現できる場合がある。さらに本章では無限長のリストに対するトレースを有限の表現で代替する方法を提案する。

5.1 正規化によるプログラムの解析

計算機プログラムと論理式との間には一種の類似性が認められる。この事実は従来から多くの文献によって指摘されている (Manna [30], McCarthy [37])。これまでの研究によれば、プログラムと論理式との対応付けを考慮する場合には従来の古典論理 (classical logic) ではなく、構成的論理 (constructive logic) を用いるべきである (Constable[9], Hayashi[18], Martin-Löf[32])。

例えば Martin-Löf[32] は構成的論理に基づく代表的な論理体系を提案している。同書には表 5.1のような表が載っている。この表は論理式の形と、それに対応する証明の形を分類したものである。表 5.1の中の λ は偽 (false) を表わす命題定数、$\wedge, \vee, \supset, \forall, \exists$ は論理記号である。また証明の欄の (a,b) は a と b との組 (pair) を表わし、$i(\cdot)$ は A から直和 A+B への標準的単射、$j(\cdot)$ は B から A+B への標準的単射である。また $\lambda x.b(x)$ はラムダ式である。

論理式	証明
⊥	なし
A∧B	(a, b)
A∨B	$i(a)$ または $j(b)$
A⊃B	$\lambda x.b(x)$
$\forall x.\mathrm{B}(x)$	$\lambda x.b(x)$
$\exists x.\mathrm{B}(x)$	(a, b)

表 5.1: 構成的論理における証明の分類
(文献 [32] による)

本論文では専ら表 5.1の 2 行目を対象として考察を進める。この行は次のように読む。すなわち「論理式 A の証明が a であり、論理式 B の証明が b である時、論理式 A∧B の証明は (a,b) である」。これを [32] の記法を用いて表すと次のようになる。

$$\frac{a\in A \qquad b\in B}{(a,b)\in A\wedge B}$$

ただし本論文では ∈ の左側の部分に興味がある事が多いので、以下の記述では必ずしも [32] の記法によらず、"∈A", "∈B", "∈A∧B" などを省略した形を使う。

上の (a,b) は組 (pair) である。組を基本的な構成要素として持つプログラミング言語として LISP (McCarthy[38], McCarthy[39], Steel[51]) が良く知られている。我々は Lisp の基本的な部分のみを以下で用いる。この Lisp の部分言語を Formal LISP と名付けた。Formal Lisp も基本的には通常の LISP と同じ表記法を用いる。組 (a,b) を LISP で表すには (a.b) のようにドット (.) を含む記法を用いるのが原則である。ただし本論文では通常のプログラムと同様なデータ構造を取り扱いたいので (a b) というリスト (list) で組 (a,b) を表すこともある。厳密に言えば、リスト (a b) は (a.(b.nil)) というドット記法の略記である。

さて「リスト (a b) が A∧B の証明を表すものである」という観点で、リストに関する規則を考察する。まず、AND(∧) を含む論理式に対しては、図 5.1のような AND(∧) の推論規則が知られている。図 5.1の推論規則は自然演繹法 (Prawitz[43]) の定式化を用いている。すなわち第2章の図 2.3の中から ∧ に関する部分を抜き出したものである。∧I, ∧E はそれぞれ AND Introduction および AND Elimination の略である。すなわち Introduction の規則により論理記号 (∧) が導入され、Elimination の規則はそれを除去する。

$$\frac{A \quad B}{A\wedge B}\ \wedge\mathrm{I} \qquad \frac{A\wedge B}{A}\ \wedge\mathrm{E} \qquad \frac{A\wedge B}{B}\ \wedge\mathrm{E}$$

図 5.1: AND(∧) の論理的な推論規則 (自然演繹法の一部)

論理式の推論規則に対応して、図 5.2に示すようにリストに対する推論規則を作ることができる。cons, car, cdr は LISP で伝統的に用いられている関数の名前である。図 5.1と図 5.2とを見比べると、∧I と cons とが対応し、同様に ∧E と car, cdr とが対応していることが分かる。すなわち AND(∧) の推論規則とリストに対する規則は全く同じ形をしている。なお以下の本文ではドット (.) を用いないリスト表記を多用するので図 5.2には対応する推論規則の形も掲げておく。

$$\frac{a \quad b}{(a.b)}\ \mathtt{cons} \qquad \frac{(a.b)}{a}\ \mathtt{car} \qquad \frac{(a.b)}{b}\ \mathtt{cdr}$$

$$\frac{a \quad (b)}{(a\ b)}\ \mathtt{cons} \qquad \frac{(a\ b)}{a}\ \mathtt{car} \qquad \frac{(a\ b)}{(b)}\ \mathtt{cdr}$$

図 5.2: リストに対する推論規則

なお以下の記述の中では図を見やすくするために、本来の推論規則では car と cdr とを次のように分けて書くべきところを一つの「分解」の規則の適用のようにまとめて書くことがある。

$$\frac{(a\ b\ c)}{a}\ \mathtt{car} \qquad \frac{(a\ b\ c)}{(b\ c)}\ \mathtt{cdr} \qquad \frac{(a\ b\ c)}{a \quad (b\ c)}\ \text{分解}$$

本論文では ∧ を含む論理式とリストとが同一の規則に従うことを利用して、論理式の証明図における正規化の性質をリストの証明図に応用する。ここに正規化という処理は、証明図の中の冗長な箇所を取り除くリダクション (reduction) という操作を繰り返して行うことである (Prawitz[43], Troelstra[57])。

定義 (再掲) それ以上リダクションが適用できない証明図を**正規形**であるという。

今考えている例では、論理記号は ∧ だけで推論規則は ∧I と ∧E だけである。この論理体系における冗長な証明の形は図 5.3および図 5.4に示すものである。

$$\frac{\dfrac{A \quad B}{A\wedge B}\ \wedge\mathrm{I}}{A}\ \wedge\mathrm{E} \qquad \frac{\dfrac{A \quad B}{A\wedge B}\ \wedge\mathrm{I}}{B}\ \wedge\mathrm{E}$$

図 5.3: 冗長な証明図 (その 1)

$$\dfrac{\dfrac{A \wedge B}{A \quad B}\ \text{分解}}{A \wedge B}\ \wedge I$$

図 5.4: 冗長な証明図 (その 2)

図 5.3の証明図 (二つある) は上側の ∧I の推論規則で折角 A∧B を作ったにもかかわらず、すぐにその下で ∧E を適用して ∧ を除去している。このような証明は回りくどい。図 5.3の左の証明図を単に「A」に簡約化 (reduce) しても証明図の結論は変わらない。同様に、右の証明図は「B」に簡約化される。このように、連続した ∧I-∧E の推論規則がリダクション規則によって省かれる。図 5.3に対応するリダクションを LISP の記法で表わせば、(CAR (CONS A B)) ▷ A および (CDR (CONS A B)) ▷ B である。ここに ▷ はリダクションの方向を示す。この性質は LISP の世界では良く知られている。

図 5.4の証明図も同様に冗長である。この証明図は単に A∧B に簡約化できる。これもリダクションである。ただし、このリダクション規則を論理式の証明図に対して陽に述べている文献は少ない。例外的に Prawitz の論文 [44] の中に説明があるが、そこでの規則の使われ方は本論文とは逆の方向であり、証明図の中の minimal formula の complexity を小さくするための "expansion" の規則として用いられている。いずれにしても、図 5.4のリダクション規則はタイプ付きラムダ計算の世界では図 5.3と同様に扱われており、これらのリダクション規則に関して本論文で扱う証明図が strongly normalizable であることが保証されている。(文献 [44] では strongly normalizable を bounded と呼んでいるので注意。) 本論文では図 5.3と図 5.4のリダクションを区別せずに用いる。

図 5.4に対応する LISP の表現は (CONS (CAR (CONS A B))(CDR (CONS A B))) ▷ (CONS A B) となる。この性質も LISP の世界では良く知られている。

本論文ではプログラムの実行結果のトレースを Formal LISP で表現する。そこでは証明図の正規形に対応してトレースの正規形というものを考えることができる。正規なトレースはプログラムの実行結果の標準形 (正規形) と考えることができる。次節ではトレースの正規形の性質を利用して二つのプログラムの同値性を証明する。次節で示すのは特定の入力に対する同値性である。これを一般化するための考察を 5.3節で行う。そこでは従来の数学的帰納法を陽に使わないプログラムの拡張法を提案する。5.4節ではリスト以外のデータ構造に対して本論文の方法を適用する。

5.2 トレース情報によるプログラムの表現

前述のリストの例を続けよう。Boyer と Moore の本 [4] には次の問題が載っている。下の表現は LISP の関数 APPEND が結合的 (associative) であることを意味している。これを証明せよ、というのが問題である。

```
(EQUAL (APPEND (APPEND A B) C)
       (APPEND A (APPEND B C)) )
```

この問題を我々の方法で解いてみよう。そのためには、先ず関数 APPEND の定義を見る必要がある。下の定義の中の LISTP は Boyer と Moore の本 [4] で用いられている述語で、引数がリストであるか否かを判定する。

```
(APPEND X Y)
= (IF (LISTP X)
      (CONS (CAR X) (APPEND (CDR X) Y))
      Y)
```

Formal LISP に組み込まれている基本的な関数は car, cdr, cons だけであるから、この APPEND の定義をそのまま Formal LISP で表現する訳にはいかない。しかし、プログラムの実行結果のトレースを記述することはできる。今 A が (a b c)、B が (d e f) であるとして (APPEND A B) がどのように計算されるかを追跡してみよう。この様子が図 5.5に示されている。すなわち、入力データの (a b c) は a と (b c) とに分解される。(b c) に対しては APPEND が再帰的に適用される。最後に a と再帰的な結果とが cons によって結合される。図 5.5の内容を具体的に記述し、さらに C に対してデータ (g h i) を与えると、A と B とを先に append するプログラム (APPEND (APPEND A B) C) の実行結果のトレースは図 5.6のようになる。(以後、プログラムの実行結果のトレースを表わす Formal LISP の表現を証明図あるいは単にトレースと呼ぶ。)

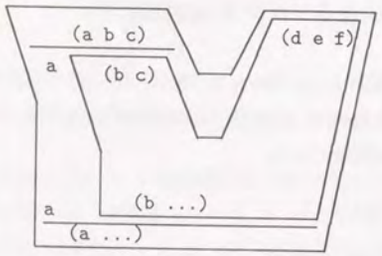

図 5.5: APPEND のトレースを作る過程

```
 (a b c)
a  (b c)
   b  (c)
      nil  c      (d e f)3
   ↓
   b          (c d e f)2
↓
a  (b c d e f)
 (a b c d e f)1
a  (b c d e f)1
   b  (c d e f)2
      c  (d e f)3
         d  (e f)
            e  (f)
               nil  f    (g h i)
            ↓
            e       (f g h i)
         ↓
         d  (e f g h i)
      ↓
      c  (d e f g h i)
   ↓
   b  (c d e f g h i)
↓
a  (b c d e f g h i)
 (a b c d e f g h i)
```

図 5.6: (APPEND (APPEND A B) C)

図 5.6には証明図を見やすくするために、付加情報が書き込んである。まずa, bの上に矢印(↓)が付いているものは証明図の上の方に出ているa, bを指している。(必ずしも直上とは限らない。)矢印で結ばれている二つのaは本来は上方のaだけ書けば良いが、証明図を読みやすくするために下方に再掲してある。さらに推論規則の横棒に数字(1, 2, 3)が付いているのは本文でリダクションの説明をする時の参照用である。

図 5.6のリダクションは次のように行われる。まず1という数字の付いた二つの推論規則は、上がconsで下がcarとcdrとをまとめた分解の規則である。これはリダクションの対象である「∧I-∧Eの連続」に該当する。この二つの推論規則1の適用を省くと,(b c d e f)の所で証明図の上と下とがうまく重なる。それと同時に「aの矢印」と「a」とがキャンセルされた形となり、図 5.7の証明図が得られる。以下同様に2,3とマークされた推論規則が順番にリダクションされる。最終的に正規形となった結果を図 5.8に示しておく。図 5.8の中には当然ながら「∧I-∧Eの連続」は現れない。

```
 (a b c)
a  (b c)
   b  (c)
      nil  c      (d e f)3
   ↓
   b          (c d e f)2
 (b c d e f)
b  (c d e f)2
   c  (d e f)3
      d  (e f)
         e  (f)
            nil  f    (g h i)
         ↓
         e       (f g h i)
      ↓
      d  (e f g h i)
   ↓
   c  (d e f g h i)
↓
b  (c d e f g h i)
↓
a  (b c d e f g h i)
 (a b c d e f g h i)
```

図 5.7: 最初のリダクションが終ったところ

```
                    (d e f)
                    d  (e f)
                       e  (f)
                          nil f      (g h i)
(a b c)                 ↓
a  (b c)                e       (f g h i)
   b  (c)            ↓ d  (e f g h i)
      nil c             (d e f g h i)
   ↓
   b        (c d e f g h i)
↓
a  (b c d e f g h i)
(a b c d e f g h i)
```

図 5.8: 正規形となった証明図

次に B と C とを先に append するプログラム (APPEND A (APPEND B C)) を考える。まず (APPEND B C) のトレースを上の図 5.6と同様に作る。次に図 5.9の結果に (a b c) を APPEND して最終的な結果を得る。この最終的な証明図は、先の正規形となった図 5.8と全く同じ証明図になる。したがってリダクションをするまでもなく、最初から正規形となっている。

```
(d e f)
d  (e f)
   e  (f)
      nil f      (g h i)
   ↓
   e       (f g h i)
↓ d  (e f g h i)
(d e f g h i)
```

図 5.9: (APPEND A (APPEND B C)) のトレース (その 1)


```
(a b c)
a  (b c)
   b  (c)
      nil c    (d e f g h i)
   ↓
   b        (c d e f g h i)
↓
a  (b c d e f g h i)
(a b c d e f g h i)
```

図 5.10: (APPEND A (APPEND B C)) のトレース (その 2)

ここで二つのプログラムのトレース (証明図) が正規化によって等しくなることの意味を考察する。まずリダクションの規則は証明図の結論 (conclusion, 最下部の論理式あるいはリスト) を変えないことが分かる。したがって、正規化の結果が等しい時には必ず結論も等しい。証明図の結論はプログラムで言えば最終的な値に相当する。結局、トレースが正規化によって等しくなる時には、二つのプログラムの出力する値が等しいばかりでなく、その途中の計算の過程 (トレース) も標準形に直せば等しくなる。これは通常のプログラムの同値の概念よりも強い性質である。

定義 2.

1. 二つのプログラム $P_1(x)$ と $P_2(x)$ のトレース $P_1(a)$ と $P_2(a)$ が正規化によって等しくなる時、この二つのプログラムは a **において t- 同値** (trace も含めて同値) であると言う。

2. 二つのプログラム $P_1(x)$ と $P_2(x)$ のトレースが、すべての x において t- 同値である時、単に **t- 同値**であると言う。

結局 APPEND の問題に対しては、(APPEND (APPEND A B) C) と (APPEND A (APPEND (B C)) とが (a b c), (d e f), (g h i) において t- 同値であることが分かった。ここでのリダクションを用いた証明は、Boyer と Moore の同値性の証明に比べると極めて簡単である。しかし、本論文の結果は与えられたデータ (a b c), (d e f), (g h i) における t- 同値を示したに過ぎない。一方 Boyer と Moore の証明は、すべての入力に対して二つのプログラムの値が等しくなることを示している。したがって、本論文の証明法で Boyer と Moore の方法を直ちに置き換えることはできない。

実際、この問題はさらに考察する価値がある。確かにリストの証明図 (図 5.6など) は特定の入力に対してのトレースを示したものである。しかし証明として表現された情報は、単に計算の結果だけでなく、計算の途中経過をも表示している。人間ならば、この情報を利用して一般の入力に対するトレースを構成することができるであろう。我々も次節で一般化の手法を考える。

なお論理式の証明図に関しては次の結果が知られている。

定理 7　証明図は木の形に表される。木の最上部の葉に相当する論理式から最下部の木の根 (*root*) に相当する論理式を結ぶ線上に並ぶ論理式の列を *spine* と言う。正規 (*normal*) な証明図の *spine* は次の 3 つの部分に分かれる。一番上の部分は、論理式を分解する推論規則 (*elimination rule*) が連続する部分である。中間の部分は分解された論理式に対する基本操作が行われる部分である。最下部は論理式を合成する推論規則 (*introduction rule*) が連続する部分である。

この定理はリストの証明図にも応用される。その結果, リストのプログラムのトレースは一種の標準形を持つことが分かる (図 5.11)。

図 5.11: 証明図の標準形

実際、図 5.8あるいは付録 A.5の図 A.3のように normalize された証明図を観察すると、上の方には分解 (elimination) に相当する car や cdr が連続して適用され、下の方では合成 (introduction) に相当する cons が専ら適用されていることが分かる。これは偶然にそうなったのではなく、上述の定理で保証されている結果である。なお付録 A.5の図 A.6も normal な証明図であるが、各部分が縮退している。

ここでは割愛するが、論理式における基本操作の例は $\wedge$ 規則の適用、等号を用いた置換、自然数論における原始帰納的述語の適用などである。リストの証明図における基本操作は本節には現れていない。なお本論文では AND に関する推論規則だけを考慮しているので、spine の定義が本来の形に比べて大幅に簡略化されている。正確な定義は Troelstra[57] の p.299 を参照されたい。

5.3　有限のトレースによる近似

一般的に言えば、トレースはプログラムの動作についての情報の一部を提供するに過ぎない。なぜならプログラムに対する「予想される入力」は一般には無限個ある。トレースだけでプログラムの挙動の全体を表現しようとすれば、無限個のトレースの集合を想定しなければならない。

本節では、プログラムがある性質を満たすときには、無限個のトレースの代りに一個のトレースでプログラム全体の挙動が表現できることを示す。ただし、その一個のトレースは概念的に「無限のリスト」に対するトレースに相当する。

最初にリストの長さを調整するために T という特別な定数を導入する。ここに T というのは cons に対する単位元 (unit) の意味である。実際には左単位元であることを用いる。

$$(\mathtt{T}\ \mathtt{x}) \equiv \mathtt{x}$$

この関係を活用してなるべく少ないトレースで多くの情報を表現することを考える。

まず次の性質を確認しておこう。いまリスト (a b c) と (d e f) に対する APPEND のトレースが存在すれば、リスト (x y z) と (u v w) に対する APPEND のトレースを**名前の付け換え** (renaming) によって得ることができる。したがって、前節の t- 同値の結果は一般的に長さ 3 のリストについて示されていると言って良い。

$$(\mathtt{a}\ \mathtt{b}\ \mathtt{c}) \leftrightarrow (\mathtt{x}\ \mathtt{y}\ \mathtt{z})$$

本当に問題となるのは、リストの長さが異なる場合である。例えば長さ 3 のリストに対するトレースを長さ 4 のリストに対するトレースで兼用する方法を考えてみよう。そのためには、入力となるリストの長さの調整が必要となる。

長さの調整 (その 1)

リスト (a b c) はリスト (x y z w) の部分リストである。

この考え方は自然である。すなわち、リスト (x y z w) を car と cdr で分解すればアトム x とリスト (y z w) を得る。後者のリスト (y z w) は長さが 3 であるから、先の注意のように renaming で (a b c) と同一のものと見なせる。

この時、アトム x を無視するために単位元 T を用いる。すなわち、T を用いて上の見方を表現すると、(x y z w) の x に T を代入したものが (a b c) と同じになる。(T y z w) $\simeq$ (a b c)

ここで別の見方を示しておくことが重要である。本論文の特徴は二つの異なる見方を立体写真 (stereoscope) のように重ね合せる点にある。

長さの調整 (その 2)

リスト (a b c) はリスト (x y z w) の先頭部分である。

これは一見奇妙であるが、実は有用な見方である。この時には (x y z w) の w に T を代入すれば良い。(x y z T) ≃ (a b c)

このように二つの異なる見方を定数 T の代入先の相違として説明出来た。同様の考察を続けると、T の代入先は x と w には限らない。下のリストはいずれも renaming をすれば (a b c) と同じリストである。T が cons 演算の単位元であることに注意。

(T y z w)
(x T z w)
(x y T w)
(x y z T)

換言すれば、上の 4 つのリストは (T x)≡x と renaming という同値関係の下では同一のリストを意味する。逆の見方をすれば、長さ 3 のリストに要素 T を補えば見かけ上の長さを 4 にすることができる。

上の説明では長さが 3 のリストと 4 のリストに終始したが、同じ議論が任意の有限長のリストに拡張出来る。さらに無限長のリスト L_ω を想定すると、任意の有限長のリストは L_ω の適当な部分に有限個の要素を代入し、他の部分には T を代入することによって得られる。別の見方をすれば、有限長のリストに T という要素を水増しすれば無限長のリストを得る。ここに ω は超準解析でいうところの non-standard な整数である (Goto[74], Martin-Löf[34])。リストの長さを表すのに non-standard な ω を用いることは既に 4.3節で論じた。

$$L_\omega = (\mathrm{a}_1\ \mathrm{a}_2 \ldots \mathrm{a}_i \ldots \mathrm{a}_j \ldots \mathrm{a}_\omega)$$
$$(\mathrm{a}\ \ \mathrm{b}\ \ \mathrm{c}) = (\mathrm{T}\ \ \mathrm{a}\ \ldots \mathrm{b}\ \ldots \mathrm{c}\ \ \ldots \mathrm{T}\ )$$

以上はプログラムの入力となるリストの長さの調整であった。このようなリストの上の同値関係がトレースにどの様に反映されるであろうか。

T を含むトレースを議論する上で重要な性質は、T が cons の左単位元であると言うことで、T を含んだトレースに対しては次のようなリダクション規則が適用される。

T を含むリダクション

(1) $\dfrac{(\mathrm{T\ a\ b\ c})}{\mathrm{T}\quad(\mathrm{a\ b\ c})}$ 分解 → (a b c)

(2) $\dfrac{\mathrm{T}\quad(\mathrm{a\ b\ c})}{(\mathrm{T\ a\ b\ c})}$ cons → (a b c)

いずれのリダクションにおいても単独で現れる T は消去されることになる。また同時に上側のリダクションにおいては (T a b c) の上方に現れるリスト中の T も消去される。また下側の (T a b c) の下方に現れるリスト中の T も同様である。これは、トレース中の T が分解や cons の対象にならない部分では、単に T を消去すれば良いことを意味する。

プログラムの入力のリスト中のある要素に T を代入して上記のリダクション規則を適用すれば、長さの短いリストに対するトレースが出来上がる。その一回り小さいトレースは T を代入する相手となる要素の数だけ得られるが、下の条件を満たせば一意に定まる。

定義 15 リスト L=(v_0 ... v_n) に対するプログラム P(L) が L に関して一様 (*uniform*) と呼ばれるのは、次の二通りのトレースがすべての i について同型になる時である。

1. 予め入力となるリストに代入 L{$\mathrm{v}_i \leftarrow$ T}, $0 \le i \le n$ を施して一回り小さなリストに対するトレースを作ったもの。
2. 元のサイズのトレースに代入 P(L){$\mathrm{v}_i \leftarrow$ T} を施し、次に T に関するリダクションを適用したもの。

ここにトレースが同型であるとは、変数の renaming および T を含むリストの同値関係により同一の形と見なせることを言う。

上の (1) で作られる一回り小さなリストは、すべての i に関して同型である。したがって、一様なプログラムではすべての i に関して (2) の結果が同型である。次に見るように APPEND や REVERSE は一様なプログラムであることが示される。

定理 8 帰納的関数 f が次の様に線形 (*linear*) に定義されている時、

$$\begin{cases} f((v_0, v_1, \ldots, v_n), y) = g(v_0, f((v_1, \ldots, v_n), h(v_0, y))) \\ f(nil, y) = e(y) \end{cases}$$

定数 T が関数 g および h の左単位元であり、しかも g と h が次の η 条件を満たすならば、f は一様な関数である。ここに η 条件とは変数の *renaming* によって関数の値が変わらないことを言う。$\lambda x.g(x,y) = \lambda z.g(z,y)$ および $\lambda x.h(x,y) = \lambda z.h(z,y)$,

Proof 正確な証明は、入力のリストの長さ n に関する数学的帰納法で行う。以下では簡単な例示により説明する。

入力のリスト $(v_0, v_1, \ldots, v_n)$ が長さ 3 のリスト (a,b,c) の場合を考える。f の値を計算すると $g(a, g(b, g(c, e(h(c, h(b, h(a,y))))))))$ となる。もし a に T を代入すると入力のリストは (b,c) となる。このリストに対する f の値は $g(b, g(c, e(h(c, h(b,y)))))$ である。一方 (T, b, c) に対する f の値は $g(\mathrm{T}, g(b, g(c, e(h(c, h(b, h(\mathrm{T}, y)))))))$ である。この二つの結果は等しい。さらに変数の renaming により $g(a, g(b, e(h(b, h(a,y)))))$ および $g(a, g(c, e(h(c, h(a,y)))))$ とも等しい。これは f が一様であることを長さ 3 のリストについて示したことになる。

(proof 終)

系 1 帰納的関数 f が次の様に tail recursive に定義されているものとする。

$$\begin{cases} f(cons(a,y)) &= h_1(a, f(y)) \\ f(nil) &= h_2 \end{cases}$$

この時、T が関数 h_1 の左単位元, $h_1(\mathrm{T}, y) = y$ であり、しかも関数 h_1 が第 1 引数について η 条件, $\lambda x.h_1(x,y) = \lambda z.h_1(z,y)$, を満たすならば、関数 f は一様な関数である。

Proof. 上の定理 8 で $h(x,y) = y$ とすれば良い。

系 2 APPEND および REVERSE は一様なプログラムである。

系 3 証明図 $P(\ldots, \mathrm{L}, \ldots)$ がリスト L に関して一様であり、証明図 $Q(\ldots, \mathrm{M}, \ldots)$ がリスト M に関して一様である時、証明図 P の結論 (最下部のリスト) がリスト M と同じ形をしていれば、二つの証明図の合成により新たな証明図 $Q(\ldots, P(\ldots, \mathrm{L}, \ldots), \ldots)$ を作ることができる。この新しい証明図はリスト L に関して一様である。

Proof $\mathrm{L}=(\mathrm{v}_0 \ \ldots \ \mathrm{v}_n)$ とする。合成された証明図 $Q(\ldots, P(\ldots, \mathrm{L}\{\mathrm{v}_i \leftarrow \mathrm{T}\}, \ldots), \ldots)$ の部分証明図 $P(\ldots, \mathrm{L}\{\mathrm{v}_i \leftarrow \mathrm{T}\}, \ldots)$ は P についての仮定によりすべて同型である。この時、各部分証明図の結論のリストも互いに同型になる。したがって Q についての仮定により新しい証明図は L に関して一様である。 (proof 終)

このように各プログラム (証明図) について一様性の証明を一度済ませておけば、後はプログラムを合成して大きなプログラムを構成することができる。本論文で扱ったプログラムの例題は、結局 APPEND あるいは REVERSE の合成であった。なお一様性の証明自身は上に示したように数学的帰納法を用いることに注意しておこう。すなわち本論文の方法を用いても数学的帰納法が全く不要になるわけではない。我々の方法は、問題を解くのに先立って各プログラムの一様性を示しておき、後は簡単な正規化の処理で同値性を証明するものである。

なお一様な証明図 (プログラム) は、LISP のプログラミングでは MAPCAR 等の関数を使って書かれることが多い。MAPCAR 等は一般の再帰的なプログラミング技法に比べて表現力が弱いが、バグが生じにくいとされている。これは一様なプログラムが素直な構造をしているからであろう。

一様なプログラム (関数) の応用を考える上では、次の性質が使われる。この性質は玉木 [55] によって最初に提案された後、我々が線形帰納的関数をカバーするために若干の拡大を行った。

定理 9 一様なプログラム $P(\mathrm{L})$ の入力リスト $\mathrm{L} = (v_0, v_1, \ldots, v_i, \ldots, v_j, \ldots)$ の中の二つの要素 v_i と v_j, $i < j$ がプログラム P の出力の中にも出現すると仮定する。この時、出力中における v_i と v_j の出現順序は次のいずれかである。

1. 正順: $\ldots v_i \ldots v_j \ldots$
2. 逆順: $\ldots v_j \ldots v_i \ldots$
3. 両方: $\ldots v_i \ldots v_j \ldots v_j \ldots v_i \ldots$, または $\ldots v_j \ldots v_i \ldots v_i \ldots v_j \ldots$ 等。

Proof 入力リスト中の要素 v_i が出力の中にも出現するならば、必ず他の要素も出現する。出現しなければ一様な関数にならない。出力中の要素の順序は、入力の順序を保存するか逆順になる。しかし要素が permutate されることはない。これを確かめるために入力リストの中の 3 つの要素 v_i, v_j, v_k, $i < j < k$ を考える。仮定により P は一様であるから、次の二つの縮退したプログラムは同型である。

1. $\mathrm{P}(\mathrm{T} \ldots \mathrm{T}, v_i, \mathrm{T} \ldots \mathrm{T}, v_j, \mathrm{T} \ldots \mathrm{T})$
2. $\mathrm{P}(\mathrm{T} \ldots \mathrm{T}, v_j, \mathrm{T} \ldots \mathrm{T}, v_k, \mathrm{T} \ldots \mathrm{T})$

もし出力中の要素の順序が $v_i \ldots v_j$ となっているとすると、v_i を v_j に rename し、v_j を v_k に rename して考えることにより三番目の要素 v_k が v_j の右にしか出現しないことが分かる。他の場合も同様に考察できる。

(proof 終)

一様なプログラムに関しては、長さ 4 のリストに関するトレースに T を代入することによって長さ 3 のリストに関するトレースを得ることができる。しかも T の代入位置によらず結果は同一である。このように長いリストに関する一様なトレースで短いリストのトレースを代用することができる。同じ考え方を無限大に移行 (transfer) すると、長さ ω のリスト L_ω に対するトレースを一つ持てば任意の有限長のリストに関するトレースを代表することになる。

この様子を図示すると下のようになる。ここに $P(L)$ はリスト L を入力とするプログラム、L_n は長さ n のリストを表す。L_0 は nil である。

$$P(L_0) \leftarrow P(L_1) \leftarrow P(L_2) \leftarrow \cdots \leftarrow P(L_\omega)$$

しかしながら、長さ ω のリストに関するトレースにリダクションを適用して有限長のリストのトレースを得るためには、無限回のリダクションを行う必要がある。つまり概念的には L_ω は有効であるが、現実の計算には役に立たない。

この問題を解決するために、トレースの一様性を利用してリストの長さを伸ばすことを考える。すなわち、長さ 3 のリストに関するトレースがリダクションによって得られたものと仮定して、元の長さ 4 のトレースを復元するのである。この時に中心となる操作は、二つのトレースの間の anti-unification である。

anti-unification は文字通りユニフィケーションの逆操作である。したがって代入の逆操作 anti-substitution が使われる (Lassez[26])。次の図に示すようにリスト (T y z w) と (x y z T) との anti-unification を取るとリスト (x y z w) が得られる。(図ではドット記法を用いている。)

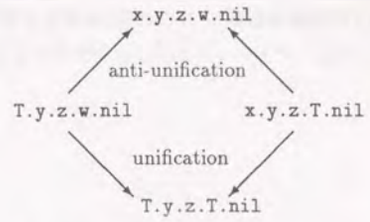

図 5.12: anti-unification と unification

問題はこのようなリスト毎の anti-unification がトレース全体に対してうまく働くかという点である。結論を言えば、答えは Yes である。下に示すアルゴリズムは小さなトレースから一回り大きなトレースを作上げるものである。

アルゴリズム

プログラム $P(L)$ が一様であるならば、トレース $P(L_n)$, $n \geq 2$, を元にして一回り大きなトレース $P(L_{n+1})$ を構成することができる。以下ではリスト L_{n+1} の各要素を表す場合には $(v_0, v_1, \ldots, v_n)$ と書く。

1. リストの各要素を rename して $P(L_n)$ のトレースから二つの同型なトレースを作る。得られるトレースは $P(v_0, v_1, \ldots, v_{n-1})$ と $P(v_1, v_2, \ldots, v_n)$ である。

2. 変数 v_0 と v_n に T を代入する。ここでは説明の都合上、二つの T を T_0 と T_n の様に区別する。代入後の二つのトレースは $P(\mathrm{T}_0, v_1, \ldots, v_{n-1})$ と $P(v_1, v_2, \ldots, v_{n-1}, \mathrm{T}_n)$ である。

3. 各トレースの中で T を消去する。その結果は、二つのトレースは $P(v_1, v_2, \ldots, v_{n-1})$ すなわち $P(L_{n-1})$ に同型となる ($P(L)$ の一様性の仮定を用いた)。ここで逆向きに考えると、大きなトレース $P(\mathrm{T}_0, v_1, \ldots, v_{n-1})$ は小さなトレース $P(L_{n-1})$ の適当な箇所に T_0 を付加することによって得られる。同様に $P(v_1, v_2, \ldots, v_{n-1}, \mathrm{T}_n)$ は $P(L_{n-1})$ に T_n を付加したものである。

4. 最後に、小さなトレース $P(L_{n-1})$ に二種類の T_0 と T_n を同時に付加する。この時 T_0 および T_n を付加する場所は、上で構成した二つのトレース $P(\mathrm{T}_0, v_1, \ldots, v_{n-1})$ およ

び $P(v_1, v_2, \ldots, v_{n-1}, \mathrm{T}_n)$ によって示される通りである。二つの T_0 と T_n の導入の間に相互干渉はない (下の定理10参照)。ここで作られた大きなトレースは anti-substitution $\{v_0 \to \mathrm{T}_0,\ v_n \to \mathrm{T}_n\}$ によってトレース $P(L_{n+1})$ と同型になる。$P(\mathrm{T}_0, v_1, \ldots, v_{n-1}, \mathrm{T}_n) \simeq P(L_{n+1})$

(アルゴリズム終)

定理 10　上のアルゴリズムにおいて T_0 の導入と T_n の導入とは独立に行うことができる。

Proof　T を含むトレースのリダクション規則の定義により、T の導入の仕方は次の 3 通りの内の何れかである。

(0) 単純な挿入: (a b c) → (T a b c), または (a b c) → (a T b c) 等の形を取るもの。

(1) 「分解」規則の追加: (a b c) → $\frac{(\mathrm{T\,a\,b\,c})}{\mathrm{T}\ \ (\mathrm{a\,b\,c})}$ のように「分解」のリダクションの逆を取るもの。

(2) cons の追加: (a b c) → $\frac{\mathrm{T}\ \ (\mathrm{a\,b\,c})}{(\mathrm{T\,a\,b\,c})}$ のように cons のリダクションの逆を取るもの。

上述のアルゴリズムの中では二つの T を T_0 と T_n の様に書き分けていた。アルゴリズムのステップ 4 における T の導入は、二つの T の各々の導入の仕方 (0), (1), (2) の組合せである。

仮定により $P(\mathrm{L})$ は一様であるから、二つの要素 v_0 と v_n の間の順序は正順あるいは逆順になる (定理 9)。したがって T_0 と T_n も正順あるいは逆順で出現する筈で、T_0 と T_n が同一の要素を表すことはない。この事実は T_0 の導入と T_n の導入との間に相互干渉がないことを示している。

(proof 終)

例題　L_4 に対する APPEND のトレースを L_3 に対する APPEND のトレースから作る。アルゴリズムのステップ 1, 2, 3 に対応するトレースを図 5.13に示す。図の中では (T_1, v_1, v_2) を (T1 y z) と書いてある。また (v_1, v_2, T_3) は (y z T3) である。アルゴリズムの最後の

ステップの様子を表 5.2にまとめておいた。この表の一番右側の欄が合成された大きなトレースに相当する。実際の一回り大きなトレースを図 5.14に示す。

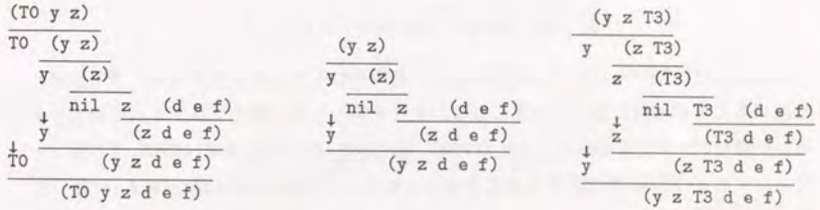

図 5.13: $P(L_n)$ から $P(L_{n-1})$ を作る過程

$(\mathrm{T}_0\ v_1\ v_2)$	→	$(v_1\ v_2)$	←	$(v_1\ v_2\ \mathrm{T}_n)$	$(\mathrm{T}_0\ v_1\ v_2\ \mathrm{T}_n)$
$\frac{(\mathrm{T}_0\ \mathrm{y\ z})}{\mathrm{T}_0\ \ (\mathrm{y\ z})}$	→	(y z)	←	$(\mathrm{y\ z\ T}_3)$	$\frac{(\mathrm{T}_0\ \mathrm{y\ z\ T}_3)}{\mathrm{T}_0\ \ (\mathrm{y\ z\ T}_3)}$
y	→	y	←	y	y
(z)	→	(z)	←	$(\mathrm{z\ T}_3)$	$(\mathrm{z\ T}_3)$
nil	→	nil	←	$\frac{(\mathrm{T}_3)}{\mathrm{nil\ T}_3}$	$\frac{(\mathrm{T}_3)}{\mathrm{nil\ T}_3}$
z	→	z	←	z	z
(d e f)	→	(d e f)	←	$\frac{\mathrm{T}_3\ \ (\mathrm{d\,e\,f})}{(\mathrm{T}_3\ \mathrm{d\,e\,f})}$	$\frac{\mathrm{T}_3\ \ (\mathrm{d\,e\,f})}{(\mathrm{T}_3\ \mathrm{d\,e\,f})}$
(z d e f)	→	(z d e f)	←	$(\mathrm{z\ T}_3\ \mathrm{d\ e\ f})$	$(\mathrm{z\ T}_3\ \mathrm{d\ e\ f})$
$\frac{\mathrm{T}_0\ \ (\mathrm{y\,z\,d\,e\,f})}{(\mathrm{T}_0\ \mathrm{y\,z\,d\,e\,f})}$	→	(y z d e f)	←	$(\mathrm{y\ z\ T}_3\ \mathrm{d\ e\ f})$	$\frac{\mathrm{T}_0\ \ (\mathrm{y\,z\,T}_3\,\mathrm{d\,e\,f})}{(\mathrm{T}_0\ \mathrm{y\,z\,T}_3\,\mathrm{d\,e\,f})}$

表 5.2: T の消去 (左側) と T の導入 (右側)

```
(T0 y z T3)
T0   (y z T3)
     y    (z T3)
          z    (T3)
               nil T3     (d e f)
          ↓
          z         (T3 d e f)
     ↓
     y         (z T3 d e f)
↓
T0        (y z T3 d e f)
     (T0 y z T3 d e f)
```

図 5.14: (T0 y z T3) に対するトレース

このように拡張アルゴリズムを用いると、有限長のリストに対するトレースを拡張して用いることが可能になる。ただし拡張が意味を持つのは一様なプログラムの場合である。一様なプログラムの例としては APPEND, REVERSE, FLATTEN, MCFLATTEN, その他ソーティングなどの Lisp の関数やその組み合わせによって合成される関数がある (後藤 [78], Goto[79], [81], [84])。

なお Böhm[3] は高階のラムダ表現を用いて我々と同様の結果を得ている。本論文の方法と彼らの方法との比較を 6.4節に述べる。

5.4 適用領域の拡大

以上のリストに対する技法は他のデータ構造にも拡張できる。簡単な方から先に議論しよう。自然数を表 5.3のようにシンボルとして表すことにより、一種のリストとして扱うことができる。

0	0
1	s.0
2	s.s.0
:	:
n	s.s...s.0
:	:

表 5.3: 自然数をリストのように表す

表 5.3の手法は PROLOG 等で数字の 5 を s(s(s(s(s(0))))) とシンボリックに書くのと同じことである。このリスト表現を用いると図 5.15のように自然数の加算のトレースがリストの APPEND のトレースと同じ形に書ける。このことから加算プログラムが結合的であることが分かる。さらに一般の自然数に対する加算のトレースの拡張法が APPEND と同様に可能であることも分かる。

```
s.s.s.0
s    s.s.0
     s    s.0
          0    s     s.s.s.0
     ↓
     s          s.s.s.s.0
↓
s            s.s.s.s.s.0
          s.s.s.s.s.s.0
```

図 5.15: 加算のトレースは APPEND に類似している

次に LISP に於ける S 式 (S-expression) を取り扱う。S 式のデータ構造はリストよりも複雑である。Boyer と Moore の本 [4] には次の問題が載っている。これは S 式の端点 (木で言えば葉) を集めるプログラムで、下の FLATTEN と言うプログラムが MC.FLATTEN と言う別の効率の良いプログラムに同値であることを主張している。

```
(FLATTEN X)
= (IF (LISTP X)
      (APPEND (FLATTEN (CAR X))
              (FLATTEN (CDR X)) )
      (CONS X "NIL") )
```

```
(MC.FLATTEN X ANS)
= (IF (LISTP X)
      (MC.FLATTEN (CAR X)
                  (MC.FLATTEN (CDR X) ANS) )
      (CONS X ANS) )
```

ここでは図 5.16のように 4 点を含む S 式に適用してみる。最初は FLATTEN の定義に忠実に木を左右に分け、結果を APPEND で集めるプログラムをを実行してみる。(図 5.17の <1>)

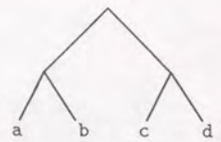

図 5.16: 簡単な S 式の例

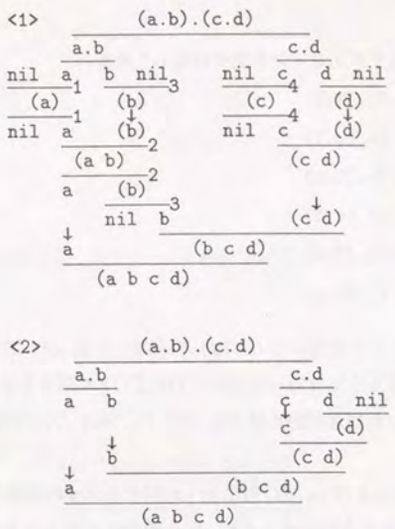

図 5.17: FLATTEN の証明図

1,2,3,4 の各推論規則をリダクションすると結果は <2> のようになる。これはまさに MC.FLATTEN のプログラムの実行結果に対応するものである。このように証明図の normalization は、S 式に対しても有効に働くことが分かる。それではリストの場合と同じ様に一般化の手法が S 式にも適用できるだろうか。

結論を先に言えば、S 式に対しても一般化の手法は応用できる。ただし anti-unification の処理はリストの場合よりも本格的になる。例題として a.b に対する FLATTEN の証明図から (a.b).(c.d) に対する証明図を構成してみる。

S 式の取り扱いは、リストの場合と基本的には共通した考え方で行える。ただし多少の拡張が必要である。例えばリストの場合には T が左単位元であるという性質を用いたが、S 式の場合には右単位元としても用いる。

単位元: $T.y \equiv y$ および $x.T \equiv x$

これに伴って T リダクション規則も拡充する必要がある。ここでは厳密な議論は省略し、例題を通して説明する。

例題

次のトレースは何れもＴリダクションを施せば等しくなる。

(1) MC.FLATTEN for (a.b).(T.T)

(2) MC.FLATTEN for (a.T).(c.T)

(3) MC.FLATTEN for (a.T).(T.d)

(4) MC.FLATTEN for (T.b).(c.T)

(5) MC.FLATTEN for (T.b).(T.d)

(6) MC.FLATTEN for (T.T).(c.d)

先のMC.FLATTENのトレースを用いて (a.T2).(c.T4) と (a.c).(T5.T7) に対する二つのトレースから一回り大きな ((a.T2).(c.T4)).(T5.T7)) に対するトレースを構成して見よう。同様な処理を続ければ最終的には ((a.T2).(c.T4)).((e.T6).(g.T8)) に対するトレースが得られる。

最初に (a.T2).(c.T4) および (a.c).(T5.T7) に対する二つの同型なトレースを作る。この二つのトレースは、図5.18に示すように、Ｔを消去すればa.cに対するトレースと同型である。次に表5.4には、図5.18の二つのトレースから一回り大きなトレースを作るための情報がまとめてある。表5.4の右側の欄にしたがってＴを導入すれば大きなトレースが得られる。この表と先のリストの場合の表5.2とを比較するとＳ式の場合の方が複雑であることが分かる。しかし、Ｔの導入自体には問題は無く、実際に大きなトレースを図5.19の様に構成することができる。

$(v_1.T_2).(v_3.T_4) \rightarrow v_1.v_3 \leftarrow (v_1.v_3).(T_5.T_7)$	$((v_1.T_2).(v_3.T_4)).(T_5.T_7)$
$(a.T_2).(c.T_4) \rightarrow a.c \leftarrow \frac{(a.c).(T_5.T_7)}{a.c \quad T_5.T_7\ldots}$	$\frac{((a.T_2).(c.T_4)).(T_5.T_7)}{(a.T_2).(c.T_4) \quad T_5.T_7\ldots}$
$\frac{a.T_2}{a \quad T_2} \rightarrow a \leftarrow a$	$\frac{a.T_2}{a \quad T_2}$
$\frac{c.T_4}{c \quad T_4} \rightarrow c \leftarrow c$	$\frac{c.T_4}{c \quad T_4}$
$\frac{T_4 \quad nil}{(T_4)} \rightarrow nil \leftarrow \frac{T_5 \quad (T_7)\ldots}{(T_5\ T_7)}$	$\frac{T_4 \quad (T_5\ T_7)\ldots}{(T_4\ T_5\ T_7)}$
$\frac{T_2 \quad (c\ T_4)}{(T_2\ c\ T_4)} \rightarrow (c) \leftarrow (c\ T_5\ T_7)$	$\frac{T_2 \quad (c\ T_4\ T_5\ T_7)}{(T_2\ c\ T_4\ T_5\ T_7)}$
$(a\ T_2\ c\ T_4) \rightarrow (a\ c) \leftarrow (a\ c\ T_5\ T_7)$	$(a\ T_2\ c\ T_4\ T_5\ T_7)$

表 5.4: Ｔの消去とＴの導入

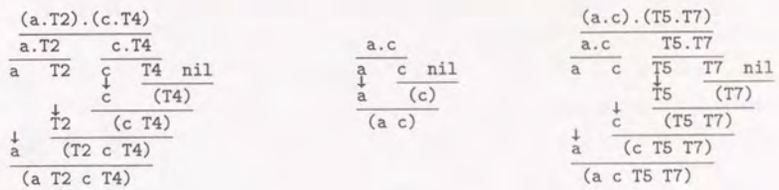

図 5.18: $P(S_n)$ のトレースから $P(S_{n-1})$ のトレースを作る

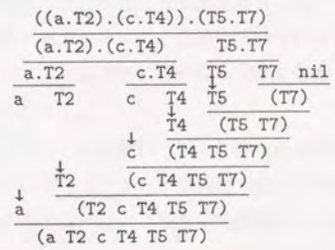

図 5.19: 得られたトレース

第 6 章

結論 — 考察・今後の研究課題

本論文では、論理的な手法によるプログラムの合成および解析の研究を述べた。この分野における従来の研究では専ら古典論理が用いられてきたが、本研究では直観主義論理を用いることにより、古典論理の場合に生じる問題を解決した。また直観主義論理を用いて具体的にプログラムを合成する方法を示し、合成の実験を行った。さらに従来は取り扱いが困難であった無限長のリストを論理的に取り扱う方法を提案し、この結果がプログラムの解析に応用できることを示した。

冒頭の序論で述べたように、本研究は「プログラムの合成と解析を自動化する」という長期的な目標に至るためのマイルストーンの役割を担っている。したがって今後の研究に残された課題も多い。本章では今後の主要な研究課題ごとに本研究で達成したことと、将来の展望・期待を述べる。

6.1 直観主義論理に基づくプログラム理論について

本研究の発端は Manna と Waldinger の論文の中に述べられていた問題である。この問題は次の様にまとめることができる。

問題: いま二つの論理式 F_1 と F_2 を考える。この二つの論理式から Manna と Waldinger の意味で二つのプログラム P_1 と P_2 がそれぞれ合成できたとする。この時、元の二つの論理式 F_1 と F_2 が論理的に同値であるとしても、出来上がった二つのプログラム P_1 と P_2 がプログラムとして同値になるとは限らない。

本論文で我々の示した解決法は、論理体系として直観主義論理を採用することである。直観主義論理は従来の古典論理から排中律を取り除いた論理体系で、推論規則の数が古典論理よりも少ない。したがって古典論理では同値と見なされる論理式でも直観主義論理においては区別できる場合がある。Manna と Waldinger が示した論理式はちょうど直観主義論理では区別できる例になっている。このようにして本論文では問題を解消した。

我々は、さらに論理式とプログラムとの対応付けを詳細に検討した。本論文では Kleene が提案した realizer の概念をプログラムの理論に応用することを提案した。realizer を用いると、直観主義論理において証明可能な論理式には帰納的関数が対応する。この事実をプログラムの理論に応用すると、異なる論理式には異なるプログラムが対応することになる。これは Manna と Waldinger の問題に対する精密な解答を与えるものである。すなわち彼らが同値であると判定した二つの論理式を realizer の観点で整理すると、一方の論理式には部分帰納的関数が対応し、他方には全域的帰納的関数が対応する。このようにして二つの論理式がプログラムとして歴然と区別される。

以上の結果が示すように、プログラムの理論に用いる論理体系としては構成的 (constructive) な特徴を持つ直観主義論理が適していることが分かった。なお本論文では特に強調しなかったが、Prolog の論理的な計算モデルを考察すると、本質的には Prolog も直観主義論理に基づいていることが分かる (後藤・古川[70], 後藤 [72])。

以下では長期的な観点に立って、論理式とプログラムとの対応関係についての一般的な問題を述べる。序論で述べた様に、論理式の証明とプログラムの間には密接な対応関係がある。表 6.1は序論の表 1.2に関数型のプログラムの記述を追加して、三つの代表的な推論規則の形とプログラムの構造との対応をまとめたものである。この関係は良く知られている (後藤 [75])。また本研究の発端となった Manna と Waldinger の論文 [29] は、この対応関係を前提として数学的帰納法とプログラムの形の詳細な分類を研究したものである。なお表 6.1に示す三つの行が、Dijkstra の提案した構造的プログラミングにおける基本構造にちょうど一致していることが容易に分かる (後藤 [71])。この事実からも論理的な証明図とプログラムとの類似性が読み取れる。

さて、本論文の第3章のように証明からプログラムを抽出する方法はアルゴリズムとして確立している。この部分を機械的に実行するのは容易である。したがって、もし仕様を表す論理式が自動的に証明できるならば、自動プログラム合成が実現する筈である (図 6.1)。

推論規則	プログラム
三段論法	逐次実行 関数の合成
場合分け	条件分岐 (if, case) 条件文 (cond, if)
数学的帰納法	ループ 再帰呼び出し

表 6.1: 推論規則とプログラムとの対応

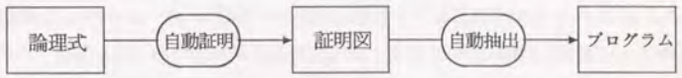

図 6.1: 自動証明とプログラムの自動合成

そこで定理の自動証明という技術に期待が集まる。確かに定理の自動証明という研究分野は存在しており、教科書も出版されている (Boyer と Moore[4], Chang と Lee[6] など)。しかし残念ながら自動的に証明できる範囲は限られている。先の表 6.1で言えば上の 2 段が自動的に証明できる範囲である。数学的帰納法の推論規則を自動化することは出来ない (図 6.2参照、この図は後藤 [83] より引用した)。もちろん数学的帰納法の一部を自動化する研究は活発に行われている。したがって図 6.2の意味は「すべての数学的帰納法の適用を自動的に行うような一般的な原理は不在である」と正しく解釈すべきである。

数学的帰納法を自動証明するための一般原理が存在しないことは、表 6.1の対応関係によればループあるいは再帰呼出を含むプログラムが自動的に合成できないことを意味する。ほとんどの実用的なプログラムはループあるいは再帰呼び出しを含むから、これを合成できなくては役に立たない。

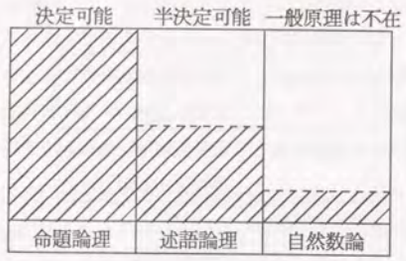

図 6.2: 自動的に定理証明のできる範囲

そこで現実的な戦略としては数学的帰納法全般を扱うのではなくて、自動的に取扱える範囲を限定して研究することである。事実、数学的帰納法の自動証明の研究はその戦略に基づいている (Boyer と Moore[4])。

我々も Prolog における帰納的なデータ構造に関して証明を行なうシステムを考察した (Goto[76])。その研究を発展させたのが、本論文の第5章で述べた一様な関数 (プログラム) の理論である。一様な関数 (プログラム) に対する正規化の処理は、本来ならば数学的帰納法を陽に適用すべき対象について、より簡単な方法で済ます工夫を述べたものである。その意味では数学的帰納法に対する一つのサブクラスを扱っている。

今後は、このような取り組みを発展させて「一般原理は不在」とされている数学的帰納法の内で、自動証明が可能なサブクラスを順次拡大する方向に研究が進むことが期待される。

6.2 直観主義論理によるプログラムの合成法について

本論文ではゲーデル解釈を応用したプログラムの合成法を提案した。ここで報告した方式は実際に Lisp のプログラムとして実現されており、証明図を与えればプログラムを生成することができる (Goto[64], [65])。なお我々の合成法を拡張して豊富なデータ構造を扱う研究も行なわれている (水上 [41], [42])。

ゲーデルが彼の解釈法を考案したのは、自然数論の無矛盾性 (consistency) を証明する目的であった (Gödel[16])。言わば純理論的な手法であったから、彼の解釈法が今日の情報工学に応用できることは極めて興味深い事実である。

なお直観主義論理を用いてプログラムを合成する方法には、本論文で示したゲーデル解釈以外の技法もある。例えば本論文の第2章で紹介した Kleene の realizer を応用する合成法も研究されている (Hayashi と Nakano[18])。ゲーデル解釈と realizer とは、両者ともに帰納的関数 (あるいは汎関数) を用いて論理式の解釈を行なう点で類似性がある。

ただし子細に比較すれば多少の違いがある。それは realizer では部分関数を用いて論理式の解釈を行なうから ($A \supset \exists z B(z)$) の形の論理式からは部分関数 (プログラム) が合成される。一方ゲーデル解釈で用いる原始帰納的汎関数は原始帰納的関数を有限のタイプに拡張したものであり、全域的 (total) である (Sato[47])。一般に全域的な関数 (プログラム) の方が扱い易い。したがって我々は帰納的汎関数に基づくゲーデル解釈を応用して研究を進めた。もちろん、論理式の性質を忠実に表すために部分関数を積極的に用いるべきだという考え方もあり得るので、 realizer の特徴が生きる場面もあると思われる。

この点に関連して、第3章の本文で少し触れた問題を扱う。ゲーデル解釈によって合成された例題の割算プログラムを走らせてみると次のようなエラーメッセージがでる。これは $7 \div 0$ すなわち除数 x_2 が 0 の場合を示している。

```
*(DIV1 7 0)
((NULL .42) T2 + NIL)
```

このエラーは LIPQ のマニュアルを参照すると「数値が期待されている箇所に非数値が来た場合」のエラーである。これは第3章の例題を子細に調べれば当然起こるエラーである。なぜなら X2 が 0 の場合には T2 の値が nil になってしまうから、この T2 の値を数値の比較の述語 LESSP の対象にはできない。

この現象を第2章の realizer の立場から見ると、このプログラムは部分関数を表す論理式から合成されたものであるから除数が 0 の場合には出力が定義されなくても良い。

しかし、ゲーデル解釈で用いた原始帰納的汎関数は原始帰納的関数を有限のタイプに拡張したものであり、上述のように全域的 (total) である。したがって合成されたプログラムは $7 \div 0$ に対しても答を返す筈である。

このように、有限のタイプの理論とプログラム合成の例題とが一致しないように見えるのが問題の内容である。この原因を調べると、第3章の割算のプログラムの合成の過程では一部の処理が理論に忠実でなかったことが分かる。具体的に言えば、最初の HA の証明図の中では幾つかの論理式を公理と見なしていた。その処置は HA の証明図を考える上では全く問題がない。しかし論理式を QT へ変換する時には注意を要する。

すなわち HA から QT への変換の定義の中では次の様に定められていた。

> 『論理式 $\mathcal{A}$ が限量記号 (quantifier)$\forall$, $\exists$ を含まない時には、$\mathcal{A}$ の中の各素論理式 $u = v$ に含まれている項 u と v とを各々ゲーデル解釈して等号で結んだものが $\mathcal{A}$ のゲーデル解釈である。』

ここで限量記号を含まないという意味は厳密に解釈しなければならない。すなわち下の論理式は例題の中で公理と見なされていた。一見したところ、この論理式には限量記号は含まれていない。しかし、比較の述語 $<$ は次の様に限量記号を用いて定義されている。したがって QT への解釈の際には限量記号を含むものとして扱う必要がある。

正確には次の様にすれば良い。まず本文では仮定として扱われていた次の論理式には限量記号が隠されていた。

$$z_2 < x_2 \supset s(z_2) < x_2 \vee s(z_2) = x_2$$

HA において最初から組み込まれている述語は $=$ であるから $<$ を $=$ を用いて定義しなければならない。

$$\exists\zeta(z_2 + s(\zeta) = x_2) \supset \exists\eta(s(z_2) + s(\eta) = x_2) \vee s(z_2) = x_2$$

同様に、仕様を表す論理式の中にも $<$ が現れていた。

$$x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2$$

これを書き直せば下の様になる。すなわち割算の商 (z_1), 剰余 (z_2) の他に ξ という変数もプログラムの出力と見なさなければならない。

$$x_1 = x_2 \cdot z_1 + z_2 \wedge \exists\xi(z_2 + s(\xi) = x_2)$$

このように論理式と証明図の解釈を正確にやり直せば正しく全域的な割算のプログラムが得られる。結果を下に示す。動作例を見れば分かるように、このプログラムは $7 \div 0$ に対しても答を返す。ただしこの答は「任意の値」を意味するから割算の答としては意味がない。

プログラムに対応する汎関数は次の様に定まる。T_1(商) と T_2(剰余) は本文の例題の通り。また T_3 は上の ξ に対応する汎関数である。

$$\begin{cases} T_1(0, x_2) &= 0 \\ T_1(s(n), x_2) &= \begin{cases} s(T_1(n, x_2) & \text{if } T_3(n, x_2) = 0 \\ T_1(n, x_2) & \text{if } T_3(n, x_2) \neq 0 \end{cases} \end{cases}$$

$$\begin{cases} T_2(0, x_2) &= 0 \\ T_2(s(n), x_2) &= \begin{cases} 0 & \text{if } T_3(n, x_2) = 0 \\ s(T_2(n, x_2)) & \text{if } T_3(n, x_2) \neq 0 \end{cases} \end{cases}$$

$$\begin{cases} T_3(0, x_2) &= pd(x_2) \\ T_3(s(n), x_2) &= \begin{cases} pd(x_2) & \text{if } T_3(n, x_2) = 0 \\ pd(T_3(n, x_2)) & \text{if } T_3(n, x_2) \neq 0 \end{cases} \end{cases}$$

Lisp によるプログラムの表現と動作例を下に示す。ここでは Lisp として KCL (湯浅・萩谷 [59]) を用いた。

```
(defun div (x1 x2) (list (t1 x1 x2) (t2 x1 x2) (t3 x1 x2)))
(defun t1 (x1 x2)
  (cond ((zerop x1) 0)
        (t (cond ((zerop (t3 (pd x1) x2)) (s (t1 (pd x1) x2)))
                 (t (t1 (pd x1) x2)) ))))
(defun t2 (x1 x2)
  (cond ((zerop x1) 0)
        (t (cond ((zerop (t3 (pd x1) x2)) 0)
                 (t (s (t2 (pd x1) x2))) ))))
(defun t3 (x1 x2)
  (cond ((zerop x1) (pd x2))
```

```
        (t (cond ((zerop (t3 (pd x1) x2)) (pd x2))
                 (t (pd (t3 (pd x1) x2))) ))))
(defun s(x) (+ 1 x))
(defun pd (x)
  (cond ((zerop x) 0)
        (t (- x 1)) ))

>(div 5 3)
(1 2 0)

>(div 6 2)
(3 0 1)

>(div 0 5)
(0 0 4)

>(div 0 0)
(0 0 0)

>(div 7 0)
(7 0 0)
```

このように本文では公理として扱っていた論理式を理論に忠実に解釈することができる。ここで示した問題の解決法は今後のプログラムの合成法に教訓として役に立つ。すなわち、プログラムを作る際にいつでもゼロからスタートする必要はない。基本的なプログラムの仕様を表す論理式を補題 (lemma) として証明しておき、そこから合成できるプログラムとともにデータベースに蓄えておく。このようなデータベースを用いると、補題を用いて証明を容易に行なうことができる。また合成されるプログラムの中には予め用意されたプログラムが含まれれる。このような方法を用いれば大規模なプログラムでも合成することが可能である。

本文で理論と例題とが一致しなかったのは、上の意味で補題とすべき論理式を公理と見なしたためにプログラムの細部が欠如したものである。

6.3 無限リストの論理的な解釈について

証明図の正規化を情報工学的に応用する研究は以前から行われていたが、本研究では正規化による計算の範囲を拡大して遅延評価 (lazy evaluation) を論理的に実現できることを示した。ここで提案した方法を用いることにより、Concurrent Prolog のようなストリーム (無限に長いリストのこと) を対象とした計算を論理的に意味づけることができる。

無限に長い要素を取り扱うのに、我々は自然数論を拡大して無限整数を導入した。この方法は超準解析 (non-standard analysis) の応用である。プログラムの論理的な取り扱いにおいては数学的帰納法が重要な役割を果たすが、数学的帰納法は無限整数の上にまで拡大できる。したがって無限整数を扱うプログラムの意味付けが可能である。

本研究では証明図の正規化を改良するために自然演繹法の $\exists$E の推論規則を Borkowski と Słpecki の推論規則 (以下では単に BS 規則と呼ぶ) で置き換えた。この新しい BS 推論規則を用いて正規化を行うと、ある条件の下では部分解を導くことができる。このような論理的な操作がコンピュータ・プログラムの世界でどのような意味を持つのか、典型的な例で考察してみよう。プログラムの仕様を表現する論理式は次の様な形を取ることが多い。

$$\forall x \exists z A(x, z)$$

ここに x は入力変数、z は出力変数である。この論理式の証明に数学的帰納法が使われるものとしよう。z に対する解を求めるためには証明図を正規化すれば良い。最初に $\forall x$ の x に入力の項 t を代入する。すると論理式が $\exists z A(t, z)$ の形になる。数学的帰納法のステップの証明では $\exists z A(a, z)$ の形の論理式を取り扱うことになる。ここで BS 規則を働かせると $\exists z$ を消すことができる。結果は $A(a, \alpha_a)$ となり、Harrop 論理式となる可能性がある。(もちろん A の中に $\vee$ あるいは $\exists$ が残っている場合は non-Harrop のままである。) $A(a, \alpha_a)$ が Harrop 論理式となれば、定理 5が適用できる可能性があり、その場合には部分解が得られる。

部分解が得られる場合には α_a という部分パラメータを含むことになる。ここに部分パラメータという考え方は Skolem 関数と似ている。事実我々の初期の発表 [73] では Skolem 関数を用いて無限項の正規化を論じていた。後に Borkowski と Słupecki の推論規則を用いた説明の方がより合理的であることが判明し、現在の記述法に変更した。

本研究で我々は自然数を拡張して無限大の整数を扱った。その方法は ω に関する次の

	公理	定理
本研究	$\omega < n$	$\omega = s(\omega \dot{-} 1)$
Martin-Löf	$\omega_k = s(\omega_{k+1})$	$s^k(0) \leq \omega$

表 6.2: 自然数の拡張法

公理を追加することであった。$\{0 < \omega,\ 1 < \omega,\ 2 < \omega, \ldots\}$. この新しい公理の下で $\omega = s(\omega \dot{-} 1)$ という等式が証明できる (定理である)。

これと若干異なる方法で Martin-Löf は自然数論を拡張した。彼の追加した公理は次の様なものである (Martin-Löf[33][34])。

$$\begin{cases} \omega_k \in N & (k = 0, 1, \ldots) \\ \omega_k = s(\omega_{k+1}) \end{cases}$$

ここに N は (拡張された) 自然数の集合を表わす。

この定式化にしたがうと、$\omega \in N$ という論理式は定義から直ちに導かれる。なぜなら定義により $\omega = \omega_0 \in N$ であるからである。このようにして $\omega_k = pd^k(\omega)$ および $\omega_k = \omega - s^k(0)$ が証明できる。なお pd という記号は predecessor function すなわち $pd(0) = 0,\ pd(s(x)) = x$ を満たす関数である。彼の体系でも $s^k(0) \leq \omega$ という関係を証明することができる。

Martin-Löf の定式化の特徴は、ω_k という一種の近似列のような考え方を用いて ω_k を $s(\omega_{k+1})$ で置き換えることである。このような操作は構成的数学の世界で choice sequence として知られているものに相当する [58]。表 6.2には我々の方法と Martin-Löf の方法を対比してまとめた。ちょうど一方では公理として選ばれている論理式が、他方では定理 (証明可能な論理式) となっていることが分かる。

本論文では無限整数ののモデル論的な側面は本格的には扱わなかった。しかし、我々の方法は超準解析の教科書に載っているウルトラフィルターの考え方を踏襲していた。Martin-Löf[33] の場合にはモデル論的な側面の議論にも及んでおり、フレッシェフィルター (Fréchet filter)[5] を採用した方が構成的論理の立場からみて適切であることを指摘している。フレッシェフィルターの名前は彼の文献には陽に登場していないが choice sequence を使った点でそれを仮定しているのである [33]。

なお注意が肝心なのは、超準解析においては「有限のもの」と「無限のもの」との区別

がつかないことである。したがって正規化の定義等も適宜修正する必要がある。なぜならリダクションのステップが「有限時間内に」終了するなどと言う表現を多用しているからである。その意味で我々の新しい正規化の手法は、正規化の手続きが有限時間内に終了するという保証はできない。リダクションのステップは第 ω 番目のステップまで行われるかも知れない。実際のコンピュータ・プログラムで正規化の手続きを実行する場合には、無限のリダクションのステップの内で最初の有限個しか得ることができない。

本論文の無限リストのテーマは二つの方向に発展する余地がある。一つは理論的な方向である。元来超準解析は微分や積分の新しい定式化として登場した理論である。したがって本論文で応用した無限大の整数は超準解析の理論の最初の一部分にしか過ぎない。さらに進んだ理論が情報工学的に見て意味があるかも知れない。Martin-Löf は解析学から物理学に及ぶ広範な影響を予測している。

もう一方では、無限リストすなわち無限整数の応用の範囲が広がると期待される。本論文の第5章では無限リストを用いたトレースの解析法を示したが、これは応用の一例である。

6.4 無限リストを応用したプログラムの解析法について

プログラムを証明図のように表示するのは特に目新しいことではない。例えばある種のプログラミング言語ではタイプ推論の説明に同様な図を用いている。本研究は単に証明図を表示するのみでなく、プログラムの実行結果(トレース)を操作する手段としてリダクションすなわち正規化を適用できると言うことを主張した。

さらに普通は数学的帰納法を用いて一般的なプログラムを表現するところを、証明図(プログラム)が一様(uniform)であるという性質を利用して陰に拡大すると言う方式を提案した。

本論文の内容を次の二つの方向に発展させることができる。一つには表5.1のAND(∧)以外の⊃, ∨を取り扱うことである。筆者の最近の研究では⊃のリダクションもプログラムのトレースとして意味を持つことが判明している(後藤[82])。また第二の方向は本文で述べた「基本操作」を含む例題に上述の手法を適用することである。基本操作には種々のバリエーションがあり得るが、我々は「置換」を基本操作としてソーティングのアルゴリズムのトレースを分析した(後藤[79])。

ところで、Böhm と Berarducci の論文[3]は我々が扱ったのと同様なプログラムの例題を二階のタイプ付きラムダ計算に基づいて分析している。ここでは二階のタイプ付きラムダ計算の効用を中心として、彼らの方法と本論文の特徴を比較してみよう。

本論文ではゲーデル解釈に伴って有限のタイプを説明した。Böhm と Berarducci の二階のタイプも基本的な考え方は同じである。ただし、二階と言う意味は関数変数を認めるということである。

例題として自然数を考える。まず普通の(タイプの無い)ラムダ計算では Church numeral として知られている表記法がある。例えば数字の「3」は次のラムダ式で表される。(後藤[77])

$$\mathbf{c}_3 \equiv \lambda fx.f^3(x)$$

つまり $\mathbf{c}_3$ は3回の反復を意味する。数字ゼロに相当するのは $\lambda fx.x$ で、これはゼロ回の反復を意味する。また successor function は受け取った numeral に具体的な変数を代入し、反復を1回増やしてからラムダ抽象化をすれば良い。すなわち $s(t) \equiv \lambda fx.f(tfx)$ となる。

Böhm と Berarducci はタイプ付きのラムダ計算を用いる。ここでは N で自然数のタイプを表すものとする。(第3章では o で自然数のタイプを表していた。)

$$\mathbf{c}_3 \equiv \lambda f^{N\to N}x^N.f^3(x)$$

さらに高階のラムダ計算であるから、タイプ変数を含む。例えば N をタイプ変数と考えることができる。しかしここではタイプ変数の説明は省略する。

このような考え方の利点は、ある種の演算を高速に行えることである。例えば3と5の加算をするには3を表すラムダ式の中の x の部分に $f^5(x)$ を代入すれば良い。また3と5を乗算するには、3の中の f の部分に f^5 を代入すれば良い。なお、このような操作が無制限に行われる訳ではなく、タイプの合致したものに限られるという意味でタイプ付きラムダ計算は安全である。

同様な考え方をリストに対しても用いることができる。すなわち、普通のリストの表記からスタートしてタイプ付きラムダ表現を導いてみると次の様になる。以下では cons を c と略記する。

$$c(a_1, c(a_2, c(a_3, nil)))$$

これを λ で抽象化すると、

$$\lambda fx.f(a_1, f(a_2, f(a_3, x)))$$

さらにタイプ(A: アトム, L: リスト)を入れると次のようになる。

$$\lambda f^{A\to(L\to L)}x^L.f(a_1, f(a_2, f(a_3, x)))$$

リストの演算に関しても自然数のように高速化がはかられる。本稿の観点から興味深いのは append の処理である。すなわちリスト $(a_1\ b_2\ c_3) = a_1.a_2.a_3.nil$ にリスト $(a_4\ b_5\ c_6) = a_4.a_5.a_6.nil$ を append するには前者の nil を後者のリストで置き換えれば良い。この append のやり方は我々の Formal Lisp の定数 T を変数と見なすことに相当する(後藤[82])。

Böhm と Berarducci の論文ではすべての関数(プログラム)を考察の対象にしている訳ではない。考察の範囲を iterative function というクラスに限って、二階ラムダ計算における表現が可能であることを証明し、プログラムの同値性に応用している。本論文では5.3節に述べた一様性の性質を重視しているので、以下にリスト構造の上の iterative function と一様な関数との比較を試みる。

結論を先に言うと、リストの上の簡単な関数は iterative でもあり一様でもある。しかし、この二つの関数のクラスは完全には一致しない。我々は iterative function に一様な関数の考え方を導入して関数のクラスを拡張することができる。

既に一様な関数の定義は述べてある。iterative function の定義を述べるためには、準備の定義としてデータ構造の記述が必要となる。なお Böhm と Berarducci の論文に忠実に紹介するのは紙面を要する。ここではリスト構造を中心として本稿で必要とする限りの informal な紹介にとどめる。

定義 16　データ構造には次の二種類がある。

1. 基本データ構造: 他のデータ構造からは生成されないもの。リスト構造の場合にはアトムの集合 A がリスト構造の例である。

2. 固有データ構造: 基本データ構造あるいは他のデータ構造から生成関数 (generator) によって生成されるデータ構造。リスト構造の場合には、リストのデータ構造 L は生成関数 $cons$: $A \times L \to L$ によって作られる。

定義 17　iterative function

基本データ構造 $A_1, A_2, \ldots, A_m$ および固有データ構造 $L_1, L_2, \ldots, L_n$ (生成関数 $g_1, g_2, \ldots, g_n$) からデータ構造 Q_i への *iterative functions* $f_i : L_i \to Q_i$ とは次の形式で定義されるものである。

$h_i : A_1 \times \cdots \times A_p \times L_1 \times \cdots \times L_q \to Q_i$

(この h_i は補助関数である。)

$f_i(g_j(a_1, \ldots, a_p, x_1, \ldots, x_q)) = h_j(a_1, \ldots, a_p, f_1(x_1), \ldots, f_q(x_q))$

iterative function の好例は APPEND である。下のように定義される関数 f を用いて $f(y,x) = append(x,y))$ と引数を反転すれば良い (Böhm と Berarducci[3])。

$$\begin{cases} f(y, cons(z,w)) &= cons(z, f(y,w)) \\ f(y, nil) &= y \end{cases}$$

APPEND のように tail recursive な関数は iterative な定義が可能である。すなわち一引数の tail recursive な関数は次の形を取る。

$$\begin{cases} f(cons(a,y)) &= h_1(a, f(y)) \\ f(nil) &= h_2 \end{cases}$$

この形の tail recursive な関数は本文の系 1で述べたように一様な関数となる。

この結果だけを眺めると、iterative function の方が一様な関数よりも一般的なクラスのような印象を受けるかも知れない。しかし、次に見るように一様な関数の考え方を用いて iterative function を拡大することができる。すなわち一様な関数も、かなり一般的な性質を規定していると言える。

関数 REVERSE は $L \to L$ の関数と考えることができる。これを iterative に定義するには次の様な形を取れば良い。

$$\begin{cases} reverse(cons(a,y)) &= h(a, reverse(y)) \\ reverse(nil) &= nil \end{cases}$$

この時、関数 h として $h(a,y) = append(y, cons(a, nil))$ とすれば確かに REVERSE が定義できる。しかし、この定義はいわゆる naive reverse の形であって効率が悪いばかりでなく、同値性の証明も難しい。証明が難しいことは Boyer と Moore[4] での REVERSE の取り扱いを見れば良く分かる。

ここで h として昔の Interlisp[56] の nconc1 が使えれば効率の良い REVERSE が定義できるのであるが、nconc1 は生成関数ではない。(ただし正確に言うと Interlisp の nconc1 は引数の順序が nconc1(list,atom) であったから、ここでの h とは逆順である。)

一方、本論文の方法では naive reverse をリダクションによって効率の良いものに最適化することが可能である (付録 A.5節参照)。結果として得られたトレースを図 6.3に示す。詳しいリダクションの過程は付録 A.5参照。

```
          (a b c)
        ---------
        (b c)   a nil
       -------  -----
       (c)  b   (a)
      ----  -------
      nil c   (b a)
            -------
            (c b a)
```

図 6.3: 正規化された REVERSE

この図 6.3を観察すると、通常の「分解」の関数とは違って cdr 部が左に、car 部が右に反転して表示されている。これは実質的に Interlisp の nconc1 の働きを内蔵していることになる。

以上の考察に基づいて、iterative function の定義を拡大する。すなわち、普通の cons によって作られるリスト L と、nconc1 を生成関数として作られるデータ構造 J の二つを

同時に考える。そして iterative function の定義は次の様にする。

$$\begin{cases} f(cons(a,y)) &= h_1(a, f(y)) \\ f(nconcl(a,y)) &= h_2(a, f(y)) \\ f(nil) &= h_3 \end{cases}$$

この定義で $h_1 = nconcl$, $h_2 = cons$ と交差して定義すれば REVERSE が得られる。しかも、その REVERSE は本研究の Formal Lisp の場合と同様に効率が良いし、証明に用いるのにも適している。

なお REVERSE は一様なプログラムであったが、ここで拡張した iterative function に関しても T が cons および nconcl の左単位元であれば一様であることが分かる。このように iterative function と Formal Lisp とを対比することにより、関数のクラスを拡張することができた。この結果は、一様な関数がかなり一般的な枠組みを提供していることを同時に示している。なお本論文で与えた一様な関数の判定法はいずれも十分条件であった。すなわち一様な関数のクラスは本論文で示した例題よりも本来は広い筈である。今後の研究によって一様な関数の性質がさらに明らかになることが期待される。

謝辞

本研究を遂行する過程で次の様に多くの方々の指導を仰いだ。ここに改めて厚く感謝する次第である。

筆者は東京大学理学部数学教室の学生として藤田宏教授 (現在東大名誉教授、明治大学教授) の指導の下に数値解析を研究していたが、藤田教授が京都大学数理解析研究所の教授を併任されていたこともあり、たびたび京都を訪れる機会を得た。京都では専門の数値解析の他にプログラム理論の分野での研究集会にも参加した。その分野の研究集会の多くは高須達教授や五十嵐滋助教授 (現在筑波大学教授) が研究代表者となっていた。学生時代に研究集会に一緒に参加した数学教室の畏友佐藤雅彦氏には、その当時からいろいろと教示を願うことがあった。

筆者は大学院修士課程を修了後、電電公社電気通信研究所に勤務することになった。同研究所では池野信一研究室長 (後に電気通信大学教授) の指導の下で幾つかの研究に従事したが、本論文で取扱ったようなプログラムの理論の研究に多くの時間を割くことができたのは幸いであった。これは池野室長のご配慮によるものである。

本論文の第 2 章に述べた realizability のテーマに関しては、直観主義論理に論理に関して小野寛晰広島大学教授 (当時津田塾大学)、佐藤雅彦東北大学教授 (当時京都大学) に教えて頂いたことが大きい。

第 3 章のゲーデル解釈に基づくプログラム合成では、Lisp を用いて実際に合成システムを作成した。筆者の研究所の先輩である竹内郁雄氏に LIPQ の使い方を教えて頂いたお蔭で、当時の日本国内では最も初期の Lisp ユーザとなることができた。

第 4 章の無限長のリストの研究は、筆者がスタンフォード大学の John McCarthy 教授のグループに滞在している間に始めた。なお研究の発表時期の点で、辛くもスウェーデンの研究に先駆けることができたのは、伊藤貴康東北大学教授にハワイのワークショップでの発表を勧められたのが勝因である。

第 5 章のトレースの取扱いは、ストックホルム大学の Per Martin-Löf 教授、ローマ大学の Corrad Böhm 教授との討論の中で形成されたアイデアである。

本論文をまとめるに当たっては、東京大学和田英一教授、井上博允教授、田中英彦教授、武市正人助教授、相田仁助教授に数々の有益なご助言を頂いた。また筆者の勤務先である NTT の戸田巌常務取締役・研究開発技術本部長、ソフトウェア研究所の塚本克治前研究部長 (現在積水化学工業応用電子研究所) には終始励まして頂いた。

参考文献

[1] Barendregt, H.P., *The Lambda Calculus, Its Syntax and Semantics, revised edition*, North-Holland, 1984.

[2] Beeson, M.J., *Foundations of Constructive Mathematics*, Springer-Verlag, 1985.

[3] Böhm, C. and Berarducci, A., Automatic Synthesis of Typed Λ-Programs on Term Algebras, *Theoretical Computer Science*, **39**, 135–154, 1985.

[4] Boyer, R.S. and Moore, J.S., *A Computational Logic*, Academic Press, 1979.

[5] Chang, C.C. and Keisler, H.J., *Model Theory (Second edition)*, North-Holland, 1977.

[6] Chang, C-L. and Lee, R.C.T., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[7] Clark, K.L. and Tärnlund, S-Å, A First Order Theory of Data and Programs, *Information Processing 77 (IFIP 77)*, 939–944, North-Holland, 1977.

[8] Constable, R.L., Constructive Mathematics and Automaticd Program Writers, *Prof. IFIP Congress 71*, 229–233, North-Holland, 1972.

[9] Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T. and Smith, S.F., *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, 1986.

[10] Diller, J., Modified Realization and the Formulae as types notion, *in To H. B. Curry: essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 491–501, Academic Press 1980.

[11] Fitting, M.C., *Intuitionistic Logic Model Theory and Forcing*, North-Holland, 1969.

[12] Floyd, R.W., Assigning Meanings to Programs, Proc. of a Symposium in Applied Mathematics, Vol. 19, *in Mathematical Aspects of Computer Science*, Amer. Math. Soc., 19–32, 1967.

[13] Gentzen, G., Investigations into Logical Deduction, *The Collected Papers of Gerhard Gentzen, M. E. Szabo (ed)*, 68–131, North-Holland, 1969, (the original title is, Untersuchungen über das logische Schliessen, *Mathematische Zeitschrift* **39**, 176–210 and 405–431, 1935).

[14] Gentzen, G., The Consistency of elementary Number Theory, *The Collected Papers of Gerhard Gentzen, M. E. Szabo (ed)*, 132–213, North-Holland, 1969, (the original title is, Die Widerspruchsfreiheit der reinen Zahlentheorie, *Mathematische Annalen* **112**, 493–565, 1936).

[15] Goad, C., Computational Use of the Manipulation of Formal Proofs, *PhD Thesis, Department of Computer Science*, Stanford University, 1980.

[16] Gödel, K., Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes, *Dialectica*, **12**, 280–287, 1958.

[17] Hagiya, M., A Proof Description Language and Its Reduction System, *Department of Information Science, Tech. Report 82-03*, University of Tokyo, Feb. 1982.

[18] Hayashi, S. and Nakano, H., *PX: A Computational Logic*, The MIT Press, 1988.

[19] Hayashi, S., Constructive Mathematics and Computer-assisted Reasoning Systems, *Proceedings of Heyting '88*, Prenum Press, London.

[20] Hindley, J.R., Lercher, B. and Seldin, J.P., *Introduction to Combinatory Logic*, Cambridge University Press, 1972.

[21] Hoare, C.A.R., An Axiomatic Basis for Computer Programming, *C. ACM*, Vol. 12, No. 10, 576–580, 583, Oct. 1969.

[22] Howard, W.A., The Formulae-as-Types Notion of Construction, *in To H. B. Curry: essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490, Academic Press 1980.

[23] Hughes G. E. and Cresswell, M J., *An Introduction to Modal Logic*, Methuen and Co Ltd., 1968.

[24] 五十嵐 滋 訳「プログラムの理論」 日本コンピュータ協会 1975, ([30] の邦訳).

[25] Lambek, J.L. and Scott, P.J., *Introduction to Higher Order Categorical Logic*, Cambridge University Press, 1986.

[26] Lassez, J-L., Mahaer, M.J. and Marriott, K.G., *Unification Revisited*, RC 12394 (#55630) IBM, T. J. Watson Research Center, 1986.

[27] Kleene, S.C., *Introduction to Metamathematics*, North-Holland, 1952.

[28] Kleene, S.C. and Vesley, R.E., *The Foundation of Intuitionistic Mathematics*, North-Holland, 1965.

[29] Manna, Z. and Waldinger, R.J., Toward Automatic Program Synthesis, *C. ACM*, Vol. 14, No. 3, 151–165, Mar. 1971.

[30] Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[31] Martin-Löf, P., An Intuitionistic Theory of Types: Predicative Part, *in Logic Colloquium 73*, pp. 73–118, North-Holland 1975.

[32] Martin-Löf, P., *Intuitionistic Type Theory*, Bibliopolis, Napoli, 1984.

[33] Martin-Löf, P., Intuitionistic Nonstandard Analysis: the mathematics of $s(s(s(\ldots)))$ and other infinitely proceeding (non-wellfounded) objects, *Logic Seminar at Mathematics Department, Stockholm University*, October-November, 1987.

[34] Martin-Löf, P., Nonstandard Type Theory, *Logic Colloquium '88, Padova*, p.14, August, 1988.

[35] Martin-Löf, P., Mathematics of Infinity, Посвящается памяти, Андрея Николаевича Колмогорова, (to appear).

[36] 松本 和夫「数理論理学・増補版」 共立出版 1971.

[37] McCarthy, J., Towards a Mathematical Theory of Computation, *Information Processing 62 (IFIP 62)*, 21–28, North-Holland, 1963.

[38] McCarthy, J., Recursive Functions of Symbolic Expression and their Computation by Machine, Part I, *Comm. ACM*, Vol.3 No.4, 184–195, April, 1960.

[39] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I., *LISP 1.5 Programmer's Manual*, The MIT Press, 1962.

[40] 三浦 聰・大浜 茂生・春藤 修二 共訳「様相論理入門」恒星社厚生閣 1981, ([23] の邦訳).

[41] 水上達就・宮本衛市・桃内佳雄 「直観主義自然数論に基づくプログラム合成のリスト処理への拡張」 電子通信学会, 論文誌 D, 81 年 1 月, Vol. J64-D, No.1, 54–61, 1981.

[42] 水上達就 「辞書式順序に基づく帰納法とそのアルゴリズム」 電子通信学会, 論文誌 D, 85 年 5 月, Vol. J68-D, No. 5, 1071–1078, 1985.

[43] Prawitz, D., *Natural Deduction: A Proof-Theoretical Study*, Almqvist and Wiksell, Stockholm, 1965.

[44] Prawitz, D., Ideas and Results in Proof Theory, *Proc. 2nd Scandinavian Logic Symposium*, 235–307, North-Holland, 1971.

[45] Robinson, A., *Non-Standard Analysis*, North-Holland, Amsterdam, 1966.

[46] Robinson, J.A., A Machine Oriented Logic based on the Resolution Principle, *J. ACM*, Vol. 12, 23–41, 1965.

[47] Sato, M., Towards a Mathematical Theory of Program Synthesis, *Sixth International Joint Conference on Artificial Intelligence (IJCAI-79)*, 757–762, 1979.

[48] Sato, M., Theory of Symbolic Expressions I, *Theoretical Computer Science* 22, 19–55, North-Holland, 1983.

[49] Shapiro, E., A subset of Concurrent Prolog and Its Implementation, *Technical Report TR-003*, ICOT, February, 1983.

[50] Shütte, K., *Proof Theory*, Springer-Verlag, 1977.

[51] Steel Jr., G.L., *Common LISP: The Language*, Digital Press, 1984.

[52] 竹内 外史・八杉 満利子「数学基礎論・増補版」 共立出版 1974.

[53] 竹内 彰一「論理型並列プログラミング言語 — Concurrent Prolog」コンピュータソフトウェア (ソフトウェア科学会), Vol. 1, No. 2, 25–37, July, 1984.

[54] Takeuchi, I. and Okuno, H., A List Processor on Mini-computer: LIPQ, 情報処理学会, 記号処理研究会研究報告集 昭和 50 年度 (1975).

[55] 玉木 久夫, private communications, 1990.

[56] Teitelman, W., *INTERLISP Reference Manual*, XEROX Palo Alto Research Center, 1974.

[57] Troelstra, A.S. (ed), *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, Springer Lecture Notes in Mathematics, 344, Springer-Verlag, 1973.

[58] Troelsta, A.S., Aspects of Constructive Mathematics, *in Handbook of Mathematical Logic (J. Barwise ed.)* pp. 973–1052, North-Holland, 1977.

[59] 湯浅 太一・萩谷 昌己「Common Lisp 入門」 岩波書店 1986.

[60] 後藤 滋樹「Recursive Realizability とプログラムシンセシス」電子通信学会, オートマトンと言語研究会 AL-75-74, 昭和 50 年 (1975).

[61] 後藤滋樹「プログラム・シンセシス (自動合成) の理論的側面」情報処理学会, 第 17 回全国大会 375–376, 1976.

[62] 後藤滋樹「プログラム・シンセシスのゲーデル解釈による基礎づけ」, 電子通信学会, オートマトンと言語研究会　AL77-16, 昭和 52 年 (1977).

[63] 後藤滋樹「最小数の原理を用いたプログラム合成」, 昭和 52 年度 情報処理学会 第 18 回全国大会, 講演番号 177, 1977.

[64] Goto, S., Program Synthesis through Gödel's Interpretation, *Lecture Notes in Computer Science*, Vol. 75, pp.302–325, Springer-Verlag, 1978.

[65] Goto, S., Program Synthesis from Natural Deduction Proofs, *Sixth International Joint Conference on Artificial Intelligence (IJCAI-79)*, 339–341, 1979.

[66] Goto, S., DURAL: an extended Prolog language, *Lecture Notes in Computer Science*, Vol. 147, pp.73–87, Springer-Verlag, 1980.

[67] 後藤滋樹「プログラム・シンセシス」情報処理学会誌, Vol. 22, No. 3, 218–225, 昭和 56 年 (1981).

[68] 後藤滋樹「述語型言語 DURAL の構成と特徴」情報処理学会, 記号処理研究会　18-6, 昭和 57 年 (1982).

[69] 後藤滋樹「計算機構と推論機構」日本数学会 1982 年度年会, 応用数学分科会・特別講演　予稿集 449–469, 昭和 57 年 (1982).

[70] 後藤滋樹・古川康一「論理型計算モデル」情報処理学会誌, Vol. 24, No. 2, 123–132, 昭和 58 年 (1983).

[71] 後藤滋樹「論理と論理プログラミング」コンピュータソフトウェア (ソフトウェア科学会), Vol. 1, No. 2, 昭和 59 年 (1984).

[72] 後藤滋樹「PROLOG 入門」サイエンス社 1984.

[73] Goto, S., Concurrency in Proof Normalization, *Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, 726–729, Los Angeles, 1985.

[74] Goto, S., Non-standard Normalization, *US-Japan Workshop*, Honolulu, May 1987.

[75] 後藤滋樹「演繹推論による自動プログラミング」情報処理学会誌, Vol. 28, No. 10, 1305–1311, 昭和 62 年 (1987). (この解説は、その後オーム社「自動プログラミングハンドブック」に再録された.)

[76] Goto, S., PT/IP: a Prolog Technology Induction Prover, *US-Japan AI Symposium 87*, pp.37–46, 1987.

[77] 後藤滋樹「記号処理プログラミング」岩波書店 1988.

[78] 後藤滋樹「Normal Programs and Partial Terms in Formal LISP」, ソフトウェア科学会, 論理と自然言語研究会　平成 1 年 3 月 (1989).

[79] 後藤滋樹「STEREO LISP によるソーティングアルゴリズムの分析」京都大学数理解析研究所「構成的数学によるプログラミング技法」研究集会 1989 年 3 月.

[80] Goto, S., Normal Programs and Proofs in Formal LISP, *Japan Czechoslovak Symposium on Theoretical Foundation of Knowledge Information Processing*, Praha, June 1989.

[81] Goto, S., How to Formalize Traces of Computer Programs, *Jumelage Meeting on Typed Lambda Calculi*, Edinburgh, September 1989.

[82] 後藤滋樹「証明図のリダクションを応用したプログラムの最適化手法」, ソフトウェア科学会, 合成変換研究会　平成 2 年 1 月 (1990).

[83] 後藤滋樹「人工知能と定理証明システム特集について」人工知能学会誌, Vol. 5, No. 1, 26–32, 平成 2 年 (1990).

[84] 後藤滋樹「Formal LISP におけるトレースの標準形とその応用」コンピュータソフトウェア (ソフトウェア科学会), Vol. 7, No. 1, 30–43, April 1990.

[85] Goto S., Proof Normalization with Non-standard Objects, *Theoretical Computer Science* (投稿中).

付録 A

詳細な証明および実例

A.1 realize する数の詳細

(定理 3の proof)

(a), (b), (c) ともに求める関数 (のゲーデル数) の存在を言う。

(a) 定理 2により $\forall x\{A(x) \supset \exists z B(x,z)\}$ は realizable である。定義 3を用いて求める関数 φ を順次詳細化して行く。

$\forall x\{A(x) \supset \exists z B(x,z)\}$ の $\forall x$ に注目すれば、適当な全域的帰納関数 α が存在して、$\alpha(\bar{x})$ は $\{A(\bar{x}) \supset \exists z B(\bar{x},z)\}$ を realize することが分かる。

この α を詳細化するには、$\{A(\bar{x}) \supset \exists z B(\bar{x},z)\}$ の "$\supset$" に注目して定義を参照すれば良い。部分帰納的関数 β が存在して、$\beta(a)$ は $\exists z B(\bar{x},z)$ を realize することが分かる。但し a は $A(\bar{x})$ を realize するものと仮定する。β を使って $\alpha(\bar{x}) = \beta$ と書ける。

結局、$\beta(a) = 2^{\bar{z}} \cdot 3^b$ となる。ここに b は $B(\bar{x},\bar{z})$ を realize する数である。

これらを組み合わせて、$e = \Lambda\bar{x}\Lambda a.(2^{\bar{z}} \cdot 3^b)$ が $\forall x\{A(x) \supset \exists z B(x,z)\}$ を realize する関数となる。

求める φ のゲーデル数は $\Lambda\bar{x}.\{e(\bar{x},a)\}_0$ である。ここに $\{\ \}_0$ とは p_i を i 番目の素数とする時、次式で定義される原始帰納的関数 $\{\ \}_i$ において $i = 0$ としたものである。

$$\{a\}_i = \mu x_{x<a}\{\ p_i^x \mid a \ \wedge\ \neg p_i^{s(x)} \mid a\ \}$$

ここに $p_i^x \mid a$ は「p_i^x は a を整除する」という意味である。すなわち $\{a\}_i$ とは a を素因数分解した時の p_i の指数 (ベキ乗の数) を指す。ここでは $\{2^{\bar{z}} \cdot 3^b\}_0 = \bar{z}$ として用いている。後に $\{\ \}_1$ も使う。

(b) 定理2により $\forall x\exists z\{A(x) \supset B(x,z)\}$ はrealizableである。上と同様に定義3を用いて求める関数 ψ を構成することができる。

まず全域的な帰納的関数 γ が存在して $\gamma(\bar{x})$ は $\exists z\{A(\bar{x}) \supset B(\bar{x},z)\}$ をrealizeする。

この β を詳細化するには "$\exists z$" に注目して、$\gamma(\bar{x}) = 2^{\bar{z}} \cdot 3^{c}$, ここに c は $\{A(\bar{x}) \supset B(\bar{x},\bar{z})\}$ をrealizeする数である。次に c を詳細化するには "$\supset$" に注目する。a が $A(\bar{x})$ をrealizeし、b が $B(\bar{x},\bar{z})$ をrealizeするとして $c = \Lambda a.b$ と書ける。この関数は部分帰納的関数であるから、その値は定義されない場合がある。しかし、この関数自体は $A(\bar{x})$ がrealizableであると否とにかかわらず存在することに注意しよう。但し $\bar{z}$ の方は $A(\bar{x})$ がrealizableでない場合にはどのような値になっているか分からない(任意の値で良い)。

以上をまとめると、$f = \Lambda\bar{x}.(2^{\bar{z}} \cdot 3^{\Lambda a.b})$ が $\forall x\exists z\{A(x) \supset B(x,z)\}$ をrealizeすることが分かる。求める ψ のゲーデル数は $\Lambda\bar{x}.\{f(\bar{x})\}_0$ である。ψ が全域的であることは見易い。

(c) 最後に部分帰納的関数 σ の存在を言う。そのためには補題1で下の事実が証明されていたことを思いだそう。(補題1は今の自然数論においてももちろん成り立つ。)

$\vdash \forall x\exists z\{A(x) \supset B(x,z)\} \supset \forall x\{A(x) \supset \exists z B(x,z)\}$

定理2により、$\forall x\exists z\{A(x) \supset B(x,z)\} \supset \forall x\{A(x) \supset \exists B(x,z)\}$ はrealizableである。

realizeする数を g と書く、g は $\Lambda p.q$ の形をしている。ここに p は $\forall x\exists z\{A(x) \supset B(x,z)\}$ を、q は $\forall x\{A(x) \supset \exists z B(x,z)\}$ をrealizeする数である。三たび定義3を用いれば(上に見てきたように)、g は

$$g = \Lambda p\Lambda\bar{x}\Lambda a.(2^{\{p(\bar{x})\}_0} \cdot 3^{\{p(\bar{x})\}_1(a)})$$

と書けることが分かる。この g の存在は補題1によって保証されているが、今は仮定により $\forall x\exists z\{A(x) \supset B(x,z)\}$ のrealizabilityも保証されている(realizeする数は f)ので、$g(f)$ が定義されてその値は e である。求める部分帰納的関数 σ のゲーデル数は $\Lambda\bar{z}.\{g(\Lambda\bar{x}.(2^{\bar{z}} \cdot 3^{\Lambda a.b}, \bar{x}, a)\}_0$ である。この関数 σ が部分的帰納関数であるのは、a が組み込まれているためである。性質 $\forall\bar{x}\{\varphi(\bar{x}) \simeq \sigma(\psi(\bar{x}))\}$ を確かめるのは容易。

(proof終)

A.2 プログラム合成の実例

A.2.1 割算プログラムの別解

第2章で見たように、割算プログラムの仕様を表す論理式には二通りある。ここでは本文とは異なる論理式からプログラムを合成してみる。

$\boxed{1}$ 仕様を表す論理式:

$\forall x_1\forall x_2\exists z_1\exists z_2\{x_2 \neq 0 \supset (x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}$

$\boxed{2}$ HAにおける証明図: 第3章の本文で証明された割算の論理式を定理として活用する。全体の証明は $x_2 = 0 \vee x_2 \neq 0$ を公理として用いている。この論理式には $\vee$ が使われているから後に合成されるプログラムには $x_2 = 0$ の判定が含まれる。

$$\dfrac{\dfrac{\dfrac{\overset{(\text{公理})}{x_2 = 0 \vee x_2 \neq 0} \qquad \dfrac{\overset{(\text{定理})}{\forall x_1\forall x_2\{x_2 \neq 0 \supset \exists z_1\exists z_2(x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)\}}}{x_2 \neq 0 \supset \exists z_1\exists z_2(x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}}{x_2 = 0 \vee \exists z_1\exists z_2(x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}}{\exists z_1\exists z_2(x_2 \neq 0 \supset x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}\,(*1)}{\forall x_1\forall x_2\exists z_1\exists z_2(x_2 \neq 0 \supset x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}$$

ただし(*1)は以下の二つの証明を組合せて得られる派生規則である。λ は「恒に偽」を表す記号で、$\lambda \supset A$ はどのような論理式 A に対しても成り立つ。

$$\dfrac{\dfrac{\dfrac{\overset{(\text{公理})}{x_2 = 0 \wedge x_2 \neq 0 \supset \lambda} \qquad \overset{(\text{公理})}{\lambda \supset (x_1 = x_2 \cdot t_1 + t_2 \wedge t_2 < x_2)}}{x_2 = 0 \wedge x_2 \neq 0 \supset (x_1 = x_2 \cdot t_1 + t_2 \wedge t_2 < x_2)}}{x_2 = 0 \supset (x_2 \neq 0 \supset x_1 = x_2 \cdot t_1 + t_2 \wedge t_2 < x_2)}}{x_2 = 0 \supset \exists z_1\exists z_2(x_2 \neq 0 \supset x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}$$

および

$$\dfrac{\dfrac{\dfrac{\overset{(\text{公理})}{(x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \wedge x_2 \neq 0 \supset (x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}}{(x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \supset (x_2 \neq 0 \supset x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}}{(x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \supset \exists z_1\exists z_2(x_2 \neq 0 \supset x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}}{\exists z_1\exists z_2(x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \supset \exists z_1\exists z_2(x_2 \neq 0 \supset x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}$$

$\boxed{3}$ TT において証明すべき論理式:

$x_2 \neq 0 \supset x_2 \cdot T_q(x_1, x_2) + T_r(x_1, x_2) \wedge T_r(x_1, x_2) < x_2$

T_q, T_r は TT における証明図が出来上がれば具体的に求まる。

$\boxed{4}$ TT における証明図: 本文の例題で証明された TT の論理式を定理として用いる。

$$\dfrac{\dfrac{\overset{(\text{公理})}{x_2 = 0 \vee x_2 \neq 0} \qquad \overset{(\text{公理})}{x_2 \neq 0 \supset x_1 = x_2 \cdot T_1(x_1, x_2) + T_2(x_1, x_2) \wedge T_2(x_1, x_2) < x_2}}{x_2 = 0 \vee (x_2 \neq 0 \wedge x_1 = x_2 \cdot T_1(x_1, x_2) + T_2(x_1, x_2) \wedge T_2(x_1, x_2) < x_2)}}{x_2 \neq 0 \supset x_1 = x_2 \cdot \begin{bmatrix} \text{任意} & \text{if } x_2 = 0 \\ T_1(x_1, x_2) & \text{if } x_2 \neq 0 \end{bmatrix} + \begin{bmatrix} \text{任意} & \text{if } x_2 = 0 \\ T_2(x_1, x_2) & \text{if } x_2 \neq 0 \end{bmatrix} \wedge \begin{bmatrix} \text{任意} \\ T_2(x_1, x_2) \end{bmatrix} < x_2} \; (*1')$$

ここに推論規則 (*1′) は次のようにして導かれる。

$$\dfrac{\dfrac{\overset{(\text{公理})}{x_2 = 0 \wedge x_2 \neq 0 \supset \curlywedge} \qquad \overset{(\text{公理})}{\curlywedge \supset (x_1 = x_2 \cdot t_1 + t_2 \wedge t_2 < x_2)}}{x_2 = 0 \wedge x_2 \neq 0 \supset (x_1 = x_2 \cdot t_1 + t_2 \wedge t_2 < x_2)}}{x_2 = 0 \supset (x_2 \neq 0 \supset x_1 = x_2 \cdot t_1 + t_2 \wedge t_2 < x_2)}$$

および

$$\dfrac{\overset{(\text{公理})}{(x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \wedge x_2 \neq 0 \supset (x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}}{(x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2) \wedge x_2 \neq 0 \supset (x_2 \neq 0 \supset x_1 = x_2 \cdot z_1 + z_2 \wedge z_2 < x_2)}$$

$\boxed{5}$ プログラムに対応する汎関数: T_q と T_r の定義の中では T_1 と T_2 が参照される。下の「任意」は文字通り任意の値で良い。ただし任意の値ではあっても undefined ではない。この点が本文の例題とは異なる。

$$T_q(x_1, x_2) = \begin{cases} \text{任意} & \text{if } x_2 = 0 \\ T_1(x_1, x_2) & \text{if } x_2 \neq 0 \end{cases}$$

$$T_r(x_1, x_2) = \begin{cases} \text{任意} & \text{if } x_2 = 0 \\ T_2(x_1, x_2) & \text{if } x_2 \neq 0 \end{cases}$$

$\boxed{6}$ Lisp によるプログラムの表現と動作例: 本文の例題で合成された二つのプログラム T1 と T2 を利用する形で TQ と TR が定義される。値の中の A は「任意の値」を表す。


```
(DE TQ (X1 X2) (COND ((ZEROP X2) (QUOTE A)) ((NOT (ZEROP X2)) (T1 X1 X2))))

(DE TR (X1 X2) (COND ((ZEROP X2) (QUOTE A)) ((NOT (ZEROP X2)) (T2 X1 X2))))
```

プログラムの動作例: 下の動作例の中の (EVALS DIV1) は LIPQ で先の例題のプログラムを読込んでいるところである。最後の二つの例は各々 $0 \div 0$, $7 \div 0$ を表している。このように除数 (x_2) が 0 の場合には A が出力される。

```
(DE DIV2 (X1 X2) (LIST (TQ X1 X2) (TR X1 X2)))

(EVALS DIV1)

*(DIV2 5 3)
(1 2)

*(DIV2 6 2)
(3 0)

*(DIV2 0 5)
(0 0)

*(DIV2 0 0)
(A A)

*(DIV2 7 0)
(A A)
```

A.2.2 効率の良い割算

第3章の本文の例題および上のプログラムの合成の例では、被除数 x_1 に関する数学的帰納法で HA の証明を行った。このため合成されるプログラムは、x_1 から 1 を反復して引くという能率の悪い方法で割算を実行した。ここでは証明の仕方を変えることにより、プログラムの能率を上げる。すなわち x_1 から 1 を繰り返して引く代りに、x_1 から x_2 を引けなくなるまで引く。

下の証明図の中では最小数の原理 (the least number principle) と呼ばれる論理式を用いる (後藤 [63])。この原理は自然数論における定理の一つである。その内容は「ある自然数 n について $\mathcal{A}(n)$ が成り立つならば $\mathcal{A}(x)$ を成り立たせる最小の自然数 m, $m \leq n$ が存在する」という主張である。最小数の原理を論理式で表すと下の様になる。

$$\exists x \mathcal{A}(x) \supset \exists y \{\mathcal{A}(y) \wedge \forall u (u < y \supset \neg \mathcal{A}(u))\}$$

この論理式を証明するには数学的帰納法を用いれば良い。証明は比較的簡単である (例えば Kleene の教科書 [27] に載っている)。ただし証明の過程で次の論理式を使うことに留意しておきたい。

$$\mathcal{A}(x) \vee \neg \mathcal{A}(x)$$

この論理式はいわゆる排中律 (law of the excluded middle) の形をしている。直観主義論理に基づく自然数論では排中律を無条件に認める訳にはいかない。そこで上の論理式を仮定に残して、最小数の原理を次の形で表す。

$$\frac{\mathcal{A}(x) \vee \neg \mathcal{A}(x)}{\exists x \mathcal{A}(x) \supset \exists y \{\mathcal{A}(y) \wedge \forall u (u < y \supset \neg \mathcal{A}(u))\}}$$

この形の最小数の原理を応用して割算の仕様を表す論理式が証明できる。下では $\mathcal{A}(x) \equiv x_1 < x_2 \cdot s(x)$ という論理式に対して最小数の原理が適用される。x の最小数が割算の商を表す。なお下の証明図では紙幅の都合で $x_2 \cdot s(x)$ などを乗算の記号 $(\cdot)$ を省いて $x_2 s(x)$ の様に書く。

$$\dfrac{\dfrac{\dfrac{\dfrac{x_2 \neq 0 \supset x_1 < x_2 s(x_1)}{x_2 \neq 0 \supset \exists x (x_1 < x_2 s(x))} \quad \dfrac{x_1 < x_2 s(x) \vee x_1 \geq x_2 s(x)}{\exists x (x_1 < x_2 s(x)) \supset \exists z_1 \{(x_1 < x_2 s(z_1)) \wedge \forall u (u < z_1 \supset x_1 \geq x_2 s(u))\}}}{x_2 \neq 0 \supset \exists z_1 \{(x_1 < x_2 s(z_1)) \wedge \forall u (u < z_1 \supset x_1 \geq x_2 s(u))\}}}{x_2 \neq 0 \supset \exists z_1 \exists z_2 (x_1 = x_2 z_1 + z_2 \wedge z_2 < x_2)}}{\forall x_1 \forall x_2 \{x_2 \neq 0 \supset \exists z_1 \exists z_2 (x_1 = x_2 z_1 + z_2 \wedge z_2 < x_2)\}}$$

なお上の証明図の中では次の二つの派生規則を用いた。

$$\frac{x_1 < x_2 \cdot s(z_1)}{(x_1 \dot{-} x_2 \cdot z_1) < x_2}$$

および

$$\frac{\forall u (u < z_1 \supset x_1 \geq x_2 \cdot s(u))}{x_1 = (x_1 \dot{-} x_2 \cdot z_1) + x_2 \cdot z_1}$$

派生規則の証明を詳細に書けば各々数行にわたるが、ここでは証明の細部は議論しないから割愛する。このようにして得られた HA の証明図を TT で解釈すると次の様な汎関数を得る。

まず最小数の原理自体が次のように解釈される。

$$\frac{(t_1(x) = 0 \wedge \mathcal{A}(x)) \vee (t_1(x) \neq 0 \wedge \neg \mathcal{A}(x))}{\mathcal{A}(x) \supset \{\mathcal{A}(t_2(x)) \wedge (u < t_2(x) \supset \neg \mathcal{A}(u))\}}$$

ここに $t_1(x)$ は $\mathcal{A}(x)$ が成り立てば 0, 成り立たなければ 1 となる汎関数である。$t_2(x)$ が求める商であるが、$t_2(x)$ の具体的な形を定めるためには先の HA での証明を TT で解釈する必要がある。ここでは詳しい解釈は省略し、その代りに下の各汎関数の意味を簡単に説明する。

割算の商を求めるためには、$t_1(x)$ の値を追跡すれば良い。具体的に言うと、$x_1 < x_2 \cdot s(x)$ となる最小の x の値を求める必要がある。証明図は帰納法を用いているから x は 0 からスタートする。x の値が次第に大きくなり、いずれ $x_1 < x_2 \cdot s(x)$ となる。その値を捕捉するために $t_3(x)$ という汎関数を用いる。$t_3(x)$ が 1 かつ $t_1(x)$ が 0 の時に $t_4(s(x))$ が x の値を取り、これが $t_2(x)$ の値となる。言葉で説明すると煩瑣であるが、具体例で考えると簡単な操作である。下の表は $7 \div 3$ の商が 2 であることを計算するときの汎関数の値である。この時の剰余は $x_1 - x_2 \cdot t_2(x)$ として計算される。

x	0	1	2	3	4	5
$t_1(x)$	1	1	0	0	0	0
$t_3(x)$	1	1	1	0	0	0
$t_4(x)$	any	any	any	2	2	2
$t_2(x)$	any	any	2	2	2	2

$$
\begin{aligned}
t_1(n) &= \begin{cases} 0 & \text{if } x_1 < x_2 \cdot s(x) \\ 1 & \text{if } x_1 \geq x_2 \cdot s(x) \end{cases} \\
t_2(n) &= t_4(s(n))
\end{aligned}
$$

$$
\begin{aligned}
t_3(0) &= 1 \\
t_3(s(n)) &= \begin{cases} 0 & \text{if } t_3(n) = 0 \\ \left\{ \begin{array}{ll} 0 & \text{if } t_1(n) = 0 \\ 1 & \text{if } t_1(n) = 1 \end{array} \right\} & \text{if } t_3(n) = 1 \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
t_4(0) &= \text{任意} \\
t_4(s(n)) &= \begin{cases} t_4(n) & \text{if } t_3(n) = 0 \\ \left\{ \begin{array}{ll} n & \text{if } t_1(n) = 0 \\ \text{任意} & \text{if } t_1(n) = 1 \end{array} \right\} & \text{if } t_3(n) = 1 \end{cases}
\end{aligned}
$$

これを Lisp で表現すればプログラムを得る。(省略)

A.2.3 素数を求めるプログラム

次の論理式は「いかなる x に対しても、x より大きな素数 z が存在する」という意味である。すなわち「素数というものは無限に存在する」という主張をしている。

$$\forall x \exists z (Prime(z) \wedge x < z)$$

この論理式から合成されるプログラムは x を入力すると、x よりも大きな素数 z を返す。

上の論理式を HA で証明するために、多少技巧的ではあるが、次の論理式に対して数学的帰納法を適用する (Goto[64])。ここに "$u \mid v$" とは、u が v を整除する (剰余がない) という意味である。

$\forall n \{ \exists z (z \leq s(n) \wedge Prime(z) \wedge x < z) \vee \forall y ((1 < y \wedge y \leq s(n)) \supset \neg (y \mid (x! + 1))) \}$

上の論理式の n に $x!$ を代入すれば最初の論理式が得られることに注意しよう。なお証明図の中では次の略記を用いた。

$P(n, x) \equiv \exists z (z \leq s(n) \wedge Prime(z) \wedge x < z)$

$Q(n, x) \equiv \forall y ((1 < y \wedge y \leq s(n)) \supset \neg (y \mid (x! + 1)))$

$P(n, x)$ と $Q(n, x)$ を用いると HA において証明すべき論理式は $\forall n (P(n, x) \vee Q(n, x))$ となる。

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg(1 < y \wedge y \leq s(0))}{1 < y \wedge y \leq s(0) \supset (y \mid (x! + 1))} \quad \vdots \quad Q(0, x)}{P(0, x) \vee Q(0, x)} \quad \dfrac{\dfrac{\dfrac{\dfrac{z \leq s(n) \supset z \leq s(s(n))}{\vdots \quad P(n, x) \supset P(s(n), x)}}{P(n, x) \supset P(s(n), x) \vee Q(s(n), x)}}{P(n, x) \vee Q(n, x) \supset P(s(n), x) \vee Q(s(n), x)}}{\forall n (P(n, x) \vee Q(n, x) \supset P(s(n), x) \vee Q(s(n), x))}}{\forall n (P(n, x) \vee Q(n, x))}}{Q(n, x) \supset P(s(n), x) \vee Q(s(n), x)} \quad \vdots
\qquad
\begin{array}{c} s(s(n)) \mid (x! + 1) \wedge Q(n, x) \supset P(s(n), x) \qquad \neg(s(s(n)) \mid (x! + 1)) \wedge Q(n, x) \supset Q(s(n), x) \\ \hline \end{array}
$$

合成されたプログラムの Z53 が z に対応するものである。また D71 というプログラムは $\exists d \forall n \{ (d = 0 \wedge P(n, x)) \vee (d \neq 0 \wedge Q(n, x)) \}$ の d に対応するプログラムで $P(n, x)$ が成り立つか $Q(n, x)$ が成り立つかの判定を与える。さらに #FT2 と #F84 は任意の関数であるが、これはゲーデルの公理 $A \supset (A \vee B)$ の解釈から生じるものである。

```
(DE D71 (NN X)
```

```
(COND ((ZEROP NN) 1)
      (T (COND
          ((EQ (D71 (- NN) X) 0) 0)
          ((EQ (D71 (- NN) X) 1)
           (EV (DIV (S (S (- NN))) (+ (FACT X) 1))) )))))

(DE Z53 (NN X)
 (COND ((ZEROP NN) (#F2 X))
       (T (COND
           ((EQ (D71 (- NN) X) 0) (Z53 (- NN) X))
           ((EQ (D71 (- NN) X) 1)
            (COND
             ((DIV (S (S (- NN))) (+ (FACT X) 1))
              (S (S (- NN))) )
             (T (#F84 (- NN) X)) ))))))
```

Z53 を使う際には (Z53 (FACT X) X) の形を取るが、これは上述のように n に $x!$ を代入する処理が必要であるためである。

```
*(Z53 (FACT 1) 1)
2

*(Z53 (FACT 2) 2)
3

*(Z53 (FACT 3) 3)
7

*(Z53 (FACT 4) 4)
5
```


A.3 部分パラメータの論理的な性質

部分パラメータを論理的に説明するために、証明図をタイプ付きラムダ式で解釈する。ここでは Diller[10] によるタイプ付きラムダ式の定義を採用する。この定義は Martin-Löf[31] に似ており、$a \vdash A$ という表現を用いる。この表現は「a はタイプ A の項である」という意味を持つ。この表現はまた「a はタイプ A の対象である」とか「a は論理式 A の証明である」と解釈しても良い。最後の表現は Formulae-as-types の考え方と呼ばれるものである。変数は $\lambda, \forall, \exists$ によって束縛されることがある。但し以下の定義では $\exists$ の部分のみを詳しく書く。

定義 18 タイプ付きラムダ式を次の様に定義する。

1. o はタイプである。o は自然数を表すタイプである。
2. A と B が既にタイプであることが分かっているおり、しかもタイプ A の変数 x^A が B の中のいかなる自由変数のタイプにも現れない時、$\forall x^A(B)$ と $\exists x^A(B)$ はタイプである。
3. 任意のタイプ A に対して変数 x^A はタイプ A の項の一種である。
4. $\exists$ を含む項の性質
   - ($\exists$I) $a \vdash A$ および $b \vdash B[a]$ が成り立つならば、$\langle a, b\rangle \vdash \exists x^A(B[x^A])$ である。
   - ($\exists$E) $c \vdash \exists x^A(B[x^A])$ ならば $j_1 c \vdash A$ および $j_2 c \vdash B[j_1 c]$ が成り立つ。
   - ($\exists\triangleright$) $j_1\langle a, b\rangle \triangleright a$ および $j_2\langle a, b\rangle \triangleright b$ が成り立つ。ここに $x \triangleright y$ とは「x は y にリダクション可能」と読む。

定義 19 変数 x^A が B に現れない時、$\forall x^A(B)$ を $A \supset B$ と略記する。同様に $\exists x^A(B)$ を $A \wedge B$ と略記する。

今 $c \vdash \exists x^A(B[x^A])$ という形の証明があったとする。この証明に対して BS 規則を適用すれば $b \vdash B[\alpha^A]$ という形の証明を得る。このとき部分パラメータ α のタイプは x^A のタイプと同じ筈であるからすなわち A である。このようにして $\alpha \vdash A$ および $b \vdash B[\alpha^A]$ が成り立つことが分かる。一方、定義 18の中の ($\exists$E) によれば $j_1 c \vdash A$ および $j_2 c \vdash B[j_1 c]$ が成り立つ筈である。この両者の結果を併せて考えると α と $j_1 c$ の間には何か共通する

性質があるに違いない。実際、部分パラメータ α のタイプ付ラムダ式としての正規形は j_1c の正規形と同一の形を取る。これを下のように表わす。

$$\alpha^N \equiv (j_1c)^N,$$

ここに t^N は項 t の正規形であり、これは t によって一意に決まる。この関係式が部分パラメータの論理的な性質を規定している。もし、本文で述べたような証明図の正規化の過程においてもタイプ付ラムダ式の使用が認められていたならば、項 j_1c が部分パラメータ α と同様の働きをするように記述することができる。もし部分パラメータ α を j_1c のように陽に記述すれば、それはもはや部分的な情報を表わすと言うよりも証明図の一部の情報を圧縮して表現した項と見るのが正しい。

本文の第4章で **R2′** というリダクション規則を掲げたが、**R2′** を j_1c を使って書くことができる。その場合には **R2′** 規則は次のような簡略化を行なう。すなわち $A(j_1c)$ を $A(t)$ に簡略化する。ここに $j_1c \triangleright t$ であるものと仮定する。このリダクション規則は $\Pi(j_1c)$ の形をした論理式に適用できる。すなわち **R2′** の適用対象は単一の論理式 $A(\alpha)$ には限らず、一般に部分証明 $\Pi(\alpha)$ と考えて良い。

A.4 分離された証明図の性質

(定理 6の proof)

本文で説明した分離された証明図が正しく自然演繹法の証明図になっていることを示すために、Prawitz[43] の用いた証明図の規定を用いる。

定義 20 自然演繹法の証明 (*proof*) および半証明 (*quasi-proof*) を次のように定義する。

(i) 論理式の木 (*tree*)Π が半証明であるのは、次の条件を満たす時である。Π の中に論理式 B が現れ、しかも木の中で B のすぐ上に論理式 $A_1, A_2 \ldots, A_n$ が現れるとする。この時 $(A_1, A_2 \ldots, A_n \,/\, B)$ が自然演繹法の推論規則となっている。

(ii) 半証明 Π の仮定となっている論理式の集合の上に discharge function と呼ばれる関数 $\mathcal{F}$ が定義される。$\mathcal{F}$ は論理式 A に対して A 自身あるいは A よりも木の中で下にある論理式を対応付ける。仮定 A が落ちる (*discharged*) とは $\mathcal{F}$ が存在して B に対して $\mathcal{F}(A) = B$ が成り立つことを言う。ここに $A \neq B$ と仮定する。

(iii) 半証明 Π に対する discharge 関数 $\mathcal{F}$ が正則 (*regular*) であるとは次の条件を満たすことである。

1. $\forall$I 推論規則のパラメータ a は、この $\forall$I 規則の前提が依存する仮定の中には現れない。
2. $\mathcal{F}(A)$ の値は $\vee$E 規則の C である。$\mathcal{F}(B)$ の値は $\vee$E 規則の前提 C である。$\mathcal{F}(A)$ の値は $\supset$I 規則の B である。$\mathcal{F}(A(a))$ の値は $\exists$E 規則の C である。$\mathcal{F}(A(a))$ の値は IND 規則の $A(s(a))$ である。(ここで推論規則の形は図 2.3および図 2.7を参照した。)

(iv) 半証明 Π の結論が A であり、Π の regular な discharge 関数 $\mathcal{F}$ が次の性質を満たす時、すなわち結論 A が $\mathcal{F}$ によれば仮定 Γ にのみ依存する時、Π は仮定 Γ から結論 A を導く証明になる。

定理 (再掲) 分離された証明図 $\Pi - \Pi_0$ は自然演繹法の証明図である。論理式 $Con(\Pi_0)$ は分離された証明の落ちていない仮定かあるいは自然数固有の公理になる。

Proof まず分離された証明図 $\Pi - \Pi_0$ が論理式の木の形を取ることは明らかである。次に $\Pi - \Pi_0$ に対する discharge 関数 $\mathcal{F}'$ は、元の Π に対する discharge 関数 $\mathcal{F}$ を修正して簡単に定義できる。

$$\mathcal{F}'(A) = \begin{cases} \mathcal{F}(A) & \text{if } A \text{ occurs in } \Pi - \Pi_0 \\ \text{not defined} & \text{if } A \text{ is in } \Pi_0 \\ A & \text{if } A \text{ is } Con(\Pi_0) \end{cases}$$

この新しい関数 $\mathcal{F}'$ が簡単に定義できたのは、$\Pi - \Pi_0$ の中に $\mathcal{F}(A)$ が値を持つような A が Π_0 の中に存在しないことによる。

これが保証されるのは定理の仮定によるのであって、$\mathcal{F}'(Con(\Pi_0)) = Con(\Pi_0)$ と言うことは $Con(\Pi_0)$ が落とされないことを意味する。したがって $Con(\Pi_0)$ は自然数論固有の公理か、あるいは open な $\Pi - \Pi_0$ の仮定である。 (proof 終)

A.5 トレースによる REVERSE プログラムの解析

APPEND に似た例題をもう一つ紹介しておこう。

```
(EQUAL (REVERSE (REVERSE X)) X)
```

上の問題も Boyer と Moore の本に出ている。実際の記述 [4] では X がリストであるという付帯条件 (LISTP) が課せられているが、ここではリスト以外は考慮していないため省略する。X を (a b c) としてリストの証明図を作る。まず REVERSE のトレースを正規形にしておく。図 A.2は下の素朴な REVERSE(naive reverse) の定義から作った Formal LISP のトレースである。トレースを作る過程は前出の APPEND と同様であるが、今回は APPEND が含まれている。(図 A.1参照)

```
(REVERSE X)
= (IF (LISTP X)
      (APPEND (REVERSE (CDR X)) (CONS (CAR X) NIL))
      NIL)
```

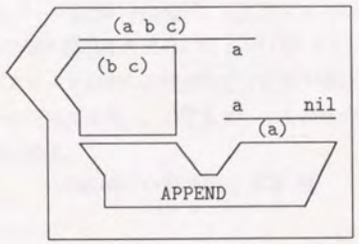

図 A.1: REVERSE のトレースを作る過程

```
                (a b c)
            ------------------
            (b c)            a
          ------------
          (c)        b
        ------
        nil c nil
            ------1   ↓
            (c)       b nil
          ------1   ------3
          nil c     (b)
                  ------2
                  (c b)
                ------2     ↓
                c   (b)     a nil
                  ------3   ------
                  nil b     (a)
                            ------
                ↓           (b a)
                c           ------
                ------------------
                  (c b a)
```

図 A.2: 素朴な REVERSE

出来上がった図 A.2は、明らかに正規形ではない。これにリダクションを適用して正規な REVERSE のトレースとしたものが図 A.3である。次に問題を解くために、REVERSE に相当する証明を二度反復した証明図 (図 A.4) を作る。この証明図も正規形ではない。なお図 A.4の中では、car と cdr をまとめた規則を APPEND の例題とは左右逆転させて用いている。

```
            (a b c)
          ------------
          (b c)      a nil
        ------     ------
        (c)   b    (a)
      ------     ------
      nil c        (b a)
           ------------
             (c b a)
```

図 A.3: 正規化後の REVERSE

リダクションの過程は前回の APPEND の証明図と似ている。ただし今回はリダクションの過程が長いので、二つに分けて説明する。まず図 A.4の中では推論規則 1，2，3 が斜めに並んでいる。これらは「∧I-∧E の連続」を表しているから、リダクションの対象である。これらを除去した中間結果の証明図の一部を左右反転して見やすく書き直したものが図 A.5である。


```
                (a b c)
            ------------
            (b c)      a   nil
          ------     ------3
          (c)   b    (a)
        ------     ------2
        nil c        (b a)
             ------------
               (c b a)
            ------1
            (b a)       c   nil
          ------2     ------
          (a)         (c)
        ------3     ------
        nil a         (b c)
             ------------
               (a b c)
```

図 A.4: REVERSE を重ねた証明図

```
            (a b c)
          ------6
          a   (b c)
            ------5
            b   (c)
              ------4
              c   nil
            ↓ ------4
            b   (c)
          ↓ ------5
          a   (b c)
          ------6
          (a b c)
```

図 A.5: 中間結果 (∧I-∧E を除去したところ)

さらに図 A.5の中の 4，5，6 は図 5.4の意味でリダクションの対象となる。これを除去すると最終的には図 A.6の様に簡単化される。図 A.6の (a b c) は、それ自身で一つの証明図である。なお、リダクション規則の適用の順番は任意で良いから、ここで説明した以外の道筋でリダクションを行っても良い。いずれにしても最終的に得られる正規な証明図は道筋によらず同一の形を取る。

(a b c)

図 A.6: 正規化した証明

上の結果は、REVERSE を反復適用するプログラムと恒等関数 X とが (a b c) において t- 同値であることを示している。しかし我々の本来の目的は単にリスト (a b c) に対してだけ REVERSE の問題を解くものではない。この課題の解決は本文に譲る。

付録 B

本研究の動機

本研究は、構成的論理を用いてコンピュータ・プログラムの合成あるいは解析を行う。このような論理的アプローチは**プログラムの理論** (Mathematical Theory of Computation) と呼ばれる。プログラムの理論の提唱は 1962 年に McCarthy が行った (McCarthy[37])。この分野の初期の研究成果としては Floyd[12] や Hoare[21] によるプログラムの正当性 (correctness) の証明が有名である。

しかし、何と言ってもプログラムの理論の研究が世界的に盛んになったのは、Manna の教科書 [30] が 1974 年に出版されてからである。この本は多くの国で翻訳されたが、中でも日本では米国で原著が出版されるのと同時に邦訳 [24] が出版されるという具合で、大いに関心が高まっていた。筆者がこの分野の研究を始めたのも、Manna の本の邦訳を担当された五十嵐滋京都大学助教授 (現在筑波大学教授) のグループに触発されて同書を読み始めた事が契機となっている。

Manna の本の第 3 章はプログラムの解析の説明に当てられていたが、プログラムの合成についての記述は 3 章末の僅か半ページの解説だけであった。筆者はプログラムの合成についての解説が少ない点にむしろ興味を抱き、Manna の本で参照されていた文献を調べ始めた。その中で最も進んでいると思われたのは Manna 自身と Waldinger の共著の論文 [29] である。彼らの論文ではプログラムの仕様 (specification) を論理式で表す。その論理式が証明可能ならば、その証明の過程の中からプログラムを抽出する。これはいわゆる論理的なプログラム合成法である。彼らの論文の主眼は、証明に用いる数学的帰納法の形が合成されるプログラムの形を規定することを示した点にある。しかし彼らの論理体系は古典論理 (classical logic) に基づいているため、仕様を表す論理式が同値であっても合成されるプログラムが同値になるとは限らない。Manna と Waldinger 自身も論文の中

で、この問題を奇妙である (原文は surprising enough) と記述している。

筆者は、この問題の原因は古典論理の推論規則が強過ぎるためであると考えた。直観主義論理は古典論理から推論規則を減らした論理体系である。検討の結果、直観主義論理を用いると Manna と Waldinger の奇妙な現象は生じないことが判明した。その他にも合成されたプログラムの実行可能性が保証されるなど、直観主義論理は好ましい性質を持つことが分かった。

本研究の他にもプログラム理論に直観主義論理を適用する提案があった。特に Constable[9] は時期的にも早く、本研究に先駆けていた。しかし Constable の論文は直観主義論理の応用の可能性を抽象的に指摘するのに留まっていた。本研究は具体的な合成の問題を解いて見せたことに意義がある。この辺の「歴史的」なサーベイは後に Hayashi[19] によって報告されている。

次に研究を進めてプログラムの合成法を研究した。具体的にはゲーデルの解釈を用いた手法を考察し、実際に Lisp を用いて合成システムを実装した。この時にゲーデル解釈を用いるべきであるという示唆は佐藤雅彦東京大学助教授 (現在東北大学教授) より頂いた。本研究の価値を強調するならば、実際にプログラムの合成システムをコンピュータの上で走らせたことである。この点が評価されたためであろうか、筆者の論文の中ではゲーデル解釈による合成システムを扱った論文 [65] が最も多く引用されている。

ゲーデル解釈の理論面については、幸いなことに日本にも専門家 (論理学者) がいた。筆者は八杉満利子静岡大学助教授 (現在は京都産業大学教授) の解説 [52] に興味を持ち、八杉助教授が京都大学数理解析研究所に短期共同研究のために滞在している間に指導を仰いだ。その際に八杉助教授が「こんな理論が本当にコンピュータのプログラムの役に立つのですか」と驚かれたことを覚えている。

筆者自身は論文 [65] を発表した前後から、ちょうどその頃日本に紹介された Prolog に興味を抱いて研究を開始した。Prolog は論理プログラミングと呼ばれていることからも想像できるように、プログラムの理論とは関係が深い。筆者は Prolog に対しても論理的な立場を取り、証明機構を二つ内蔵した Prolog の処理系の提案 [66], [68] などを行った。この Prolog の経験が後の証明図の正規化の研究に大いに役立った。

筆者は 1984 年 9 月から 1985 年 8 月まで、スタンフォード大学コンピュータ科学科に客員研究員として滞在することになった。日本出発に先立って同大学での研究内容を調査したが、その中で最も興味深かったのは Goad の博士論文 [15] である。実は彼の論文

中に Goto[65] が引用されており、それが熱心に読むことになった理由の一つである。彼の論文の中で展開されているのは証明図の正規化という手法であった。証明図の正規化は Prolog と似ており、論理式を直接に実行する。その意味では Prolog に酷似している。ただし Prolog に比べると一階述語論理を無制限に扱える反面、実行効率の面では Prolog に劣っている。このような欠点を改良すべく Goad[15] はラムダ計算を拡張した p 計算 (p-calculus) で証明を記述する方法を提案していた。p-calculus は実行効率の面で原型の正規化よりも優れている。また Goad に先立って Hagiya[17] は正規化の処理の中で計算に関係のある部分と無関係な部分とを区別する方法を提案し、無関係な部分を取り除いた正規化の記述法を与えた。

筆者はさらに正規化と Prolog との相違点を検討して、次の様な差異を明らかにした。すなわち、証明図の正規化においては証明を完了してから正規化により計算を実行するのに対して、Prolog では実行と証明とが同時に行われる。この両者は簡単な例題に対しては同様な効果を持つ。しかし計算の対象となるデータに無限長の対象が含まれている場合には、Concurrent Prolog 等ではストリームの遅延評価が可能であるのに対して、証明図の正規化では先に証明を完了しないと計算に移れない。そこで適切な論理体系を選んで無限の要素を含む証明を済ます必要がある。

この問題を解決するために、筆者は無限のデータを論理的に意味付ける理論として超準解析を用いることとし、さらに遅延評価に相当する正規化を実現するために推論規則を変形した。実際の研究の経緯は次のようであった。筆者がスタンフォード大学に滞在中に最初の論文を人工知能国際会議 (IJCAI-85) に投稿した。会議は 1985 年 8 月にロサンゼルスで開かれた。この IJCAI-85 の論文では正規化における無限要素の取り扱いを例示したものの、論理的な裏付けは不十分であった。超準解析の応用であることを明示した論文はその後 1987 年 5 月にハワイで開かれた日米ワークショップでの発表である (Goto[74])。後にこの発表の日付が意味を持つ。

というのも筆者の研究とは全く独立にスウェーデンの Martin-Löf(ストックホルム大学教授) が、やはり超準解析を応用して無限のデータに論理的な意味を持たせる研究を遂行していたのである。彼は 1987 年の 9 月から 10 月にかけてストックホルム大学の論理セミナーで講演した後、1988 年 8 月にイタリアで発表していた [33], [34]。幸いにも筆者の発表の方が先んじていたが、その差は僅かに 3ヶ月である。彼の後の論文では Goto[74] を参照してくれている (Martin-Löf[35])。

本論文の最後のテーマであるプログラムの解析法は、上記の研究が縁となり、筆者がス

ウェーデンに Martin-Löf 教授を訪ねた際に考案したものである。彼は無限要素を取り扱うために無数の無限整数 ω_k を用いている。これにヒントを得て無数の nil を持つリストを考察したのが発端である。後に形式的な表現を改良して、本論文で示すようにリストの cons 演算における単位元 T を用いる表記に改めた。この cons 演算の単位元という考え方は Hyperlisp(Sato[48]) の影響を受けている。ただし本研究では単位元を活用するのに留まっており、Hyperlisp のように Lisp 全体を atom と molecule で再構成するものではない。

筆者がトレースによる解析法 [81] をエジンバラ大学で発表した際には、ローマ大学の Böhm 教授から、彼の手法 [3] と本研究との類似性を指摘された。確かに適用できるプログラムの範囲は二つの方法に共通しているように見える。ただし Böhm の方法は二階のラムダ計算を用いているのに対して、本研究の方法はあくまでもトレースの情報を活用している。したがって、細部においては異なっている。例えば、我々の例題の中にある reverse の最適化は Böhm の方法ではうまく解けない。

筆者がこの分野の研究を開始してから 10 年以上が経過している。最初は Constable の後を追い掛ける形で研究を始めたが、後には具体的にプログラム合成のシステムを作成し、さらにはスウェーデンの研究と競うことができた。最近のトレースを利用した解析法では若干他に先行することになった。筆者がこの研究分野において多少の貢献が出来たとすれば、それはちょうど世界的な研究の進展に合わせて次々とテーマを設定できたためであろう。本研究をまとめる機会に振り返ってみると、筆者はほとんどの論文を単名で発表してきた。しかし、その研究の内容は多くの先輩友人との討論の中で育まれたものである。