ソフトウェア工学 64－9
（１９８９．２．２）

述語論理に基づく仕様から実行可能コードへの変換

小野康一† 河野誠一†† 門倉敏夫† 深澤良彰†
†早稲田大学理工学部 ††日本アイ・ビー・エム㈱

一階述語論理と集合論を基盤とする仕様を、実行可能コードに変換する手法を提案する。本手法が対象とする仕様は、述語論理式で定義するデータ間関係や状態操作を用いて、ソフトウェアの性質を表わす。変換は三つの段階からなる：述語論理式の変形で、手続き的解釈が可能な形に変形する；述語論理式から中間コードへの変換で、述語論理式の手続き的解釈にしたがって制御構造を生成する；中間コードから実行可能コードへの変換で、実行可能コードに変換する。中間コードには、λ記法に基づく関数形式を用いた。

中間コードは、その述語論理式が真となる解釈を定義域で探索する関数形式である。中間コードへの変換段階において、述語論理式に変換規則を適用し、逐次的な実行や条件分岐などの制御構造を生成する。変換規則は、述語論理式中の述語の引数についての条件などにしたがって適用される。述語の引数の情報は、述語論理式の手続き的解釈によって決定する。

実行可能コードにはLispを選び、本手法をPrologを用いてインプリメントした。

A Transformation Method from Specification based on Predicate Logic into Executable Code

Kouichi ONO†, Seiichi KAWANO††, Toshio KADOKURA†, and Yoshiaki FUKAZAWA†
† School of Science and Engineering, Waseda University, 3-4-1, Okubo Sinjuku-ku, Tokyo 160, Japan
††Yamato Laboratory-IBM Japan, Ltd. 1623-14, Shimotsuruma Yamato-shi, Kanagawa-ken 242, Japan

We propose a transformation method from a formal specification into an executable code. The specification is based on first order predicate logic and set theory. This method consists of three steps; transformation of logic formulas in order to be interpreted procedurely, transformation of logic formulas into functional form according to the procedural interpretation, and transformation from the functional expression into the executable code.

In these transformation steps, this method uses some rules to generate suitable control structures. Each rule is applied when a condition holds; for example, a condition about a binding situation for arguments of a predicate. The binding situation is decided by the procedural interpretation of the logic formula.

1．はじめに

ソフトウェア生産過程において、正確なソフトウェアを開発するための方策として、形式的仕様記述の導入がある〔1〕〔2〕。ソフトウェア開発者は、ユーザの原始要求を解析し、形式的仕様記述言語を用いて仕様を形式的に記述することで、開発対象ソフトウェアの性質や機能を正確に表現することができる。この形式的仕様記述言語のひとつに述語論理を基盤とする言語がある〔3〕。このような言語はソフトウェアの機能を論理式で記述するので、曖昧さのない、正確な記述が可能である。

一階述語論理に基づく論理プログラミングは、その立場から三つに分類できる〔4〕。

第一の立場は、論理式をもとに、その証明手順から関数プログラムを生成して、これを実行コードとする方法である〔5〕。この方法では論理式の証明の自動化が現在のところなされていないため、自動プログラミングが困難である。

第二の立場は、論理式の変形によりホーン節形式を得る方法である。この変形手法として、論理式の展開・畳み込み（fold／unfold）変換がある〔6〕。しかし、fold／unfold変換手法は自動化には至っていない。

第三の立場は、手続き的な解釈を与えることにより、論理式をプログラムとして扱う方法である〔7〕。その実行は元のプログラムの論理的演繹であり、プログラムの計算結果は元のプログラムの論理的帰結になる〔8〕。その代表的な例として、Prolog処理系があげられる。しかし、手続き的解釈を与えるために、扱う論理式を一階述語論理式の一部であるホーン節に限定している。さらに、論理式を解釈実行する機構に実行効率の点で問題があると思われる。

本手法は第二の立場に立っている。仕様中の一階述語論理式を、手続き的解釈が可能な形に変形し、その手続き的解釈にしたがって制御構造を生成し、実行可能コードへ変換する。

本手法の対象となる論理式は、全称・存在記号を用いることができるなど、ホーン節よりも記述の範囲が広い。しかし、一階述語論理によって記述されるすべての論理式を実行可能コードに変換できるわけではない。これは、変換規則の適用条件による制限のためである。しかしながら、ソフトウェアの機能を記述するための仕様としては、本手法の制限下でも十分であると考えている。

2．仕様記述言語

本手法では、仕様を以下のように定義している。

ソフトウェアの仕様
‖
データオブジェクトの定義
＋
データ操作の定義

これは、通常のプログラミング言語におけるプログラムの定義と類似している。プログラムの定義と異なるのは、データ間の関係やデータの操作を、手続き的にではなく、述語論理式を用いて宣言的に表現する、という点である。

この定義のもとで、本手法は、以下の性質を有する仕様を変換の対象とする。

2－1．オブジェクトの性質

本手法の対象とする仕様記述言語は、集合論と一階述語論理を基盤とする。対象オブジェクトの持つ性質は、それぞれの側面から、以下のように表現される。

- 集合論の側面
  - ある属性を持ったデータの集まり → 集合
- 一階述語論理の側面
  - オブジェクト中のデータ間関係 → 述語定義
  - 操作による状態遷移 → 述語論理式

また、状態遷移モデルにより、対象オブジェクトの性質を以下のように記述する。

- オブジェクトが保持している状態 → 状態変数
- 操作に対する入出力 → 入出力変数

2－2．述語の定義

データ間の関係は、述語の定義によって表現される。本手法の対象とする述語定義は、以下の記法に基づいて行なう。

$$p(x_1:S_1;\ x_2:S_2;\ \cdots\cdots\ x_n:S_n) == L$$

ただし、"=="は論理的等価（equivalent）を表わす。また、pは述語名、x_iは述語pの第i引数名、集合S_iは引数x_iの定義域、Lは一階述語論理式である。

この定義の論理的な意味は、次の通りである。

$$\forall x_1:S_1\ \forall x_2:S_2\ \cdots\cdots\ \forall x_n:S_n\ (p(x_1,x_2,\cdots\cdots,x_n)==L)$$

3．定義

変換過程の説明中で用いる用語について、以下で定義する。

3－1．論理式の木構造化

リテラルを1つのノード、and接続をノードの直列接続、or接続をノードの並列接続とみなして、論理式を木構造として表現することを考える。ただし、リテラルは、高々1個の否定演算子（"～"）がその先頭についた基本論理式（atomic formula）とする。

木構造化の例を図1aにあげる。ここで、A～Gはリテラルである。この結果の論理式をグラフとして表現すると、図1bのような木になる。論理式の木構造化は、与えられた論理式をこのような木構造を持った論理式に変形する方法である。木構造化の目的は、中間コードへの変換段階で変換規則が適用可能な形へ変形するためである。与えられた任意の論理式を木構造化できると考えているが、その証明を与えていないので、以後では、木構造化できる論理式を対象とする。

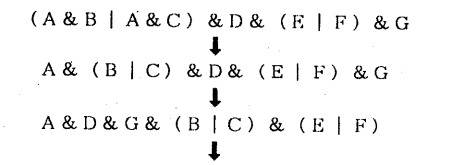


図1a．論理式の木構造化

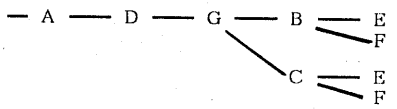


図1b．木構造化された論理式（ネットワーク表示）

3－1－1．分岐点

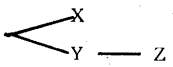
あるリテラルから下の部分木が、OR接続された複数の木からなる場合、そのリテラルを、その木の分岐点と呼ぶ。図1の木の分岐点は、リテラルG，B，Cである。

木構造化された論理式の手続き的解釈では、分岐点まで同じ状態で計算を継続し、分岐点以降の各部分木ごとに状態の切り替えを行なうことになる。

3－1－2．頭部

その木の根から最も近い分岐点までのリテラルの連言を、その木の頭部と呼ぶ。図1の木の頭部は、A＆D＆Gである。

頭部が空の場合もありうる。たとえばX｜Y＆Zという論理式は、



という木として表現され、この場合、その頭部は空である。

3－2．述語の引数のbind情報

論理式で用いられている自由変数は、状態変数または入出力変数として宣言されている。この宣言にもとづいて論理式を手続き的に解釈したとき、論理式中の述語に与えられる引数の束縛に関する情報が決定する。これを引数のbind情報と呼ぶ。

たとえば、次のような例を考える。

p(x,y) & q(y,z)

この論理式において、xが入力、y,zが出力と宣言されているとする。"&"を逐次ANDとして手続き的に解釈したとき、上の論理式における、引数のbind情報は、図2のようになる。

引数のbind情報は、論理式の変形段階で決定され、中間コードへの変換段階で用いられる。

述語	bindされている変数	bindされていない変数
p(x,y) q(y,z)	x y	y z

図2．述語の引数のbind情報

4．変換手法

4－1．変換手法の概略

本手法による変換は、以下の三つの段階からなる。

段階Ⅰ：論理式の変形

段階Ⅱ：関数形式への変換

段階Ⅲ：実行可能コードへの変換

段階Ⅰは、段階Ⅱのための前処理である。段階Ⅱが本手法による変換の主要な段階である。したがって本稿では、段階Ⅱを中心に述べる。

段階Ⅱでは、制御構造を本来持たない論理式をもとにして、制御構造を持った実行可能コードを生成するために、変換規則を適用する。変換規則には適用条件があり、条件が成立したとき、その規則に対応する制御構造が生成される。以下では、各段階における変換手法について詳しく述べる。

4－2．段階Ⅰ：論理式の変形

この段階で行なう変換は、大きく分けて次の二つである。

(1)論理式の展開

ここでは、述語定義を用いて論理式を展開し、木構造化を行なう。その目的は、以降の変換段階で関数形式に変換するために制御構造を生成可能な形に変形することと、変換により得られる関数の評価（実行）が効率的に行なわれるように論理式を簡略化することの二つである。

①制御構造のための論理式の変形

木構造化された論理式のすべての経路について調べ、後述する論理式の変形規則を適用する。その目的は、論理式を関数に変換することを考慮して、手続き的な解釈が可能な論理式を得ることである。そのために、集合演算を述語論理式と対応させて論理式を変形する。

この段階で用いる論理式の変形規則は、現時点では、集合の定義や変数の宣言、論理式中の限量記号の束縛変数の宣言に基づく変形である。

②論理式の簡略化

他の項から論理的帰結により導き出せる項の削除、矛盾する部分木の削除の二つを行なう。

(2)リテラルの順序づけ

状態変数・入出力変数の宣言と操作定義の述語論理式から、引数のbind情報が決定する。この情報に基づいて、論理式に対して逐次的な手続き的解釈を行なった際にそのすべての経路において計算可能となるように、木構造化された論理式を変形する。

論理式の論理接続が並列AND，並列ORであることから、リテラルの順序を変更する。また、その際にリテラルに対する変形規則の適用を必要とする場合には、その規則にしたがって変形する。その変換規則とは、現時点では、算術式・集合演算式の変形、補助述語の導入、無限集合を定義域とする全称記号の存在記号への変形、である。

変形により、引数のbind情報が変化するので、新たなbind情報に基づいて、リテラルの順序付けを行なう。

図３に変換の例を示す。

```
○述語定義
    p(a:integers; S:P(integers)) ==
        a>0 & a∈S   |   a≦0 & a∉S
○操作定義
    x≧0 & y≧0 & p(y-x,S)
○入出力宣言
    INPUT  x : integers; S : P(integers)
    OUTPUT y : integers
                    ⬇
○操作定義
    x≧0    &
      ( ∃n:S (y=x+n & y>x)     |
        y≧0 & y≦x & y-x∉S        )
```

図３．変換例（段階Ⅰ）

4－3．段階Ⅱ：関数形式への変換

4－3－1．変換アルゴリズム

段階Ⅰの変換によって得られた論理式を、その論理式が真となる解釈をその定義域で探索する関数形式に変換する。

引数のbind情報を参照して、木構造化された論理式を、制御構造を付加してλ記法に基づく関数形式に変換する。

論理式中の入力変数をi_i(i=1～m)とする。

論理式中の出力変数をo_j(j=1～n)とする。

操作定義の中間コードとしての関数Fは、次のように定義する。

$$F \triangleq (\lambda\, i_1\ i_2\ \cdots\ i_m.Foptree)$$

また、述語定義の中間コードとしての関数Fpは、次のように定義する。

$$Fp \triangleq (\lambda\, i_1\ i_2\ \cdots\ i_m\ f.Fptree)$$

変換アルゴリズムを疑似コードを用いて図４に表わす。

```
if (木=空)
 then
   if 論理式=操作定義
    then 次の関数形式を生成する。
            result(o1,o2,…,on)
    else /*論理式=述語定義*/ 次の関数形式を生成する。
            (λr.cond(r,false)(r))(eval(f))
 else  /* 木≠空 */
   if (木の頭部=空)
    then
      ①根に接続している複数の木の一つを選択する。
      ②次のような関数形式を生成する。
            (λr.cond(r,Frest)(r))(Ftree)
    else  /* 木の頭部≠空 */
      ①引数のbind情報に基づき、すべての引数がbind
        されているリテラルの連言を頭部から取り出す。
      ②if (リテラルの連言≠空)
         then 次のような関数形式を生成する。
                cond(Fresttree,false)(Fconj)
         else  /* リテラルの連言=空 */
           その述語に対応する変換規則を適用する。
```

図４．変換アルゴリズム

ただし図４において、中間コードの定数falseは偽を表わし、組み込み関数resultは、o_1～o_nの値を組にして返す。eval，condは、関数の評価、条件分岐を行なう。また、Foptree，Fptreeは、操作定義，述語定義の論理式を変換して得られる関数である。Ftree,Frestは、選択した木、残りの木を、それぞれ変換した関数である。Fconj，Fresttreeはそれぞれ、リテラルの連言、木の残りの部分を変換した関数である。

４－３－２．変換規則

変換規則は、以下の項目から構成される。

- 適用するリテラル
- 適用条件
- 生成する関数形式
- 生成する関数定義
- 適用時に行なう処理

以下に変換規則を分類し、上記の各項目について述べる。

(1)組み込み述語・補助述語の場合

①等号

- リテラル：
  x = exp
- 適用条件：
  x　:bindされていない変数
  exp:bindされていない変数を含まない式
- 関数形式：
  (λx.Frest)(Fexp)
- 処理　　：
  expを関数形式に変換→Fexp

ここで、Frestは、部分木の残りの部分を変換した関数形式である。

- リテラル：
  x' = exp
- 適用条件：
  x　:bindされていない状態変数
  exp:bindされていない変数を含まない式
- 関数形式：
  (λx.(λr.cond(do(update(x,x),r),false)
  (r))(Frest))(Fexp)
- 処理　　：
  expを関数形式に変換→Fexp

ここで、中間コードの組み込み関数doは、関数の評価を逐次的に行ない、updateは状態変数の更新を行なう。

②不等号

不等式の場合は、計算の停止性を保証するために、bindされていない変数についての上限と下限がともに不等式で与えられている必要がある。したがって、次のような連言について考える。

- 連言：
  $x \geqq a \ \& \ x \leqq b$
- 適用条件：
  x　:bindされていない変数
  a,b:bindされていない変数を含まない式
  $a \leqq b$
- 関数形式：
  func(Fa)
- 関数定義：
  func ≙
  (λx.cond((λr.cond(r,func(plus(x,1)))
  (r))(Frest),false)(le(x,Fb)))
- 処理　　：
  a,bを関数形式に変換→Fa,Fb

ここで、中間コードの組み込み関数plusは、加算を行ない、leは、数値の大小比較を行なう。実際には、計算途中の状態を保持するために、そのリテラルまでにbindされている変数を引数としてλ束縛しているが、説明のために本稿では略している。

③メンバーシップ

- リテラル：
  x in S
- 適用条件：
  x　:bindされていない変数
  S　:bindされていない変数を含まない式
- 関数形式：
  func(FS)
- 関数定義：
  func ≙
  (λS1.cond(false,
  (λx.(λr.cond(r,func(rest_of(S1)))
  (r))(Frest))(one_of(S1)))
  (null_set(S1)))
- 処理　　：
  Sを関数形式に変換→FS

ここで、中間コードの組み込み関数null_setは、空集合の場合に真を返し、one_of，rest_ofは、集合の１要素とそれ以外の要素の集合を返す。

④補助述語

段階Ⅰの変形によって導入される補助述語は、集合を1要素と残りの集合に分割するPsetsepaと、シーケンスを2つに分割するPseqsepaがある。

- リテラル：
  Psetsepa(S,x1,S1)
- 適用条件：
  x1,S1 :bindされていない変数
  S :bindされていない変数を含まない式
- 関数形式：
  (λS.(λx1.(λS1.Frest)
  (rest_of(S)))(one_of(S)))(FS)
- 処理 ：
  Sを関数形式に変換→FS

- リテラル：
  Pseqsepa(s,t,u)
- 適用条件：
  t,u:bindされていない変数
  s :bindされていない変数を含まない式
- 関数形式：
  func(⟨⟩,Fs)
- 関数定義：
  func ≙
  (λt u.cond(false,(λr.cond(r,
  func(add_elm(t,top(u)),tail(u)))(r))
  (Frest))(null_seq(u)))
- 処理 ：
  sを関数形式に変換→Fs

ここで、中間コードの組み込み関数add_elmは、シーケンスの後尾に一要素を追加したシーケンスを返し、topとtailは、シーケンスの先頭と残りの後部を返し、null_seqは、空シーケンスの場合に真を返す。

(2)ユーザ定義述語の場合

ユーザ定義述語の場合、その引数を入力とするか出力とするかを、定義から決定することはできない。しかし、操作定義の論理式を手続き的解釈することにより、論理式中のそれぞれのユーザ定義述語についての引数のbind情報が決定できる。

その場合、論理式中の同じユーザ定義述語が、異なる引数のbind情報で用いられるかも知れない。そのような述語に対しては、それぞれのbind情報について異なる述語として変換する。

したがって、ある一つのユーザ定義述語に対して、複数の中間コードが得られる場合がある。

- リテラル：
  $p(x_1,\cdots,x_k,y_1,\cdots,y_l)$
- 適用条件：
  $x_1,\cdots,x_k$:bindされていない変数
  $y_1,\cdots,y_l$:bindされていない変数を含まない式
- 関数形式：
  $Fp(Fy_1,\cdots,Fy_l,Frest)$
- 関数定義：
  func ≙
  述語定義の論理式を
  変換して得られる関数形式
- 処理 ：
  $y_1,\cdots,y_l$を関数形式に変換→$Fy_1,\cdots,Fy_l$

(3)限量記号により束縛されている論理式の場合

①定義域が有限集合

ア．束縛域内の論理式がbindされていない変数を含まない場合

- リテラル：
  ∀x:S hold p(x)
- 適用条件：
  S :bindされていない変数を含まない式
- 関数形式：
  func(FS)
- 関数定義：
  func ≙
  (λS1.cond(true,
  and(p(one_of(S1)),func(rest_of(S1))))
  (null_set(S1))
- 処理 ：
  Sを関数形式に変換→FS

- リテラル：
  ∃x:S hold p(x)
- 適用条件：
  S :bindされていない変数を含まない式
- 関数形式：
  func(FS)
- 関数定義：
  func ≙
  (λS1.cond(false,
  or(p(one_of(S1)),func(rest_of(S1))))
  (null_set(S1))
- 処理 ：
  Sを関数形式に変換→FS

イ．束縛域内の論理式がbindされていない変数を含む場合

段階Ｉの変換により、存在記号によって束縛されている場合だけのはずである。したがって次のような、束縛域内の論理式が真となる解釈を発見する関数を生成する。

- リテラル：
  ∃x:S hold p(x,y)
- 適用条件：
  y　:bindされていない変数
  S　:bindされていない変数を含まない式
- 関数形式：
  func(FS)
- 関数定義：
  func ≙
  (λS1.cond(false,(λx.(λr.
  cond(r,func(rest_of(S1)))(r))
  (Frest))(one_of(S1)))(null_set(S1)))
- 処理　：
  Sを関数形式に変換→FS

②定義域が無限集合

段階Ｉの変換により、存在記号によって束縛されている場合だけのはずである。

束縛域内の論理式が、その束縛変数以外に、bindされない変数を含むかどうかで、ア，イの二通りの変換に分かれる。

ア．束縛域内の論理式がbindされていない変数を含まない場合

次のような、束縛域内の論理式の充足可能性を調べる関数を生成する。

- リテラル：
  ∃x:S hold p(x)
- 適用条件：
  S　:bindされていない変数を含まない式
  （無限集合）
- 関数形式：
  cond(Frest,false)(Fp)
- 処理　：
  Sを関数形式に変換→FS

イ．束縛域内の論理式がbindされていない変数を含む場合

段階Ｉの変換により、その束縛域は木全体のはずである。したがって、その束縛変数を出力変数ではないbindされていない変数として扱い（したがってその値が結果として返ってこない）、束縛域内の木を変換する。

p(x,y)　（x,y:bindされていない）として変換

- リテラル：
  ∃x:S hold p(x,y)
- 適用条件：
  y　:bindされていない変数
  S　:bindされていない変数を含まない式
  （無限集合）
- 処理　：
  p(x,y)を
  x,y:bindされていない変数
  として変換

以上の変換規則による変換の例を、図５に示す。

```
x≧0    &
  ( ∃n:S (y=x+n & y>x)     |
    y≧0 & y≦x & y-x∉S       )
```

⬇

```
funcOP≙
 (λx S.cond((λr.cond(r,func1(0,x,S)(r))
  (func2(S,x)),false)(gt(x,0)))

func1≙
 (λy x S.cond((λr.cond(r,func1(plus(y,1)))(r))
  (cond(result(y),false)
   (member(minus(y,x),S)))),false)(le(y,x)))

func2≙
 (λS1 x.cond(false,(λn.(λr.cond(r,
  func2(rest_of(S1)))(r))
   ((λy.cond(result(y),false)(gt(y,n)))
    (plus(x,n))))(one_of(S1)))(null_set(S1)))
```

図５．変換例（段階Ⅱ）

４－４．段階Ⅲ：実行可能コードへの変換

中間コードを構成する組み込み関数を、実行可能コードの構造と対応させ、中間コードのデータ構造を実行可能コードのデータ構造と対応させることにより、実行可能コードへの変換が容易にできる。現時点では、中間コードからLispコードへ変換を行なっている。

５．検討

５－１．論理式の制限

本手法は、一階述語論理式を対象としているが、そのすべてを変換できるわけではない。以下では、論理式の制限について詳しく述べる。

述語定義の論理的な意味は、２－２で述べたように次

のようになる。

$\forall x_1 : S_1 \forall x_2 : S_2 \cdots\cdots \forall x_n : S_n \ (p(x_1, x_2, \cdots\cdots, x_n) == L)$

ここで論理的等価（"=="）を用いているが、本手法では、変換過程で展開のみを用いているので、包含（"←"）とみなすことができる。すると、述語定義はホーン節形式と類似している。ホーン節形式と異なるのは、定義の本体であるLに全称・存在記号を用いることができる、という点である。

段階Ⅱで用いる変換規則の適用条件による論理式の制限について考察する。

制限は次の通りである。

> 操作定義の論理式中のリテラルを組み込み述語になるまで展開したとき、それぞれの組み込み述語には、bindされていない変数が高々１個である（ただしシーケンスは除く）。
>
> 述語定義はこの制限を受けない。組み込み述語を組み合わせることで、ユーザ定義述語の引数にbindされていない変数を渡しても変換可能である。

Prologを比較対象として、この制限の影響について考える。

Prologでは、算術演算などの特殊な組み込み述語を除けば、可逆性が満たされている。したがって、ユーザ定義述語は（それらの組み込み述語を用いない限り）、どの引数がbindされていてもかまわない。つまり、述語の可逆性が、組み込み述語に依存するという点で、本手法の制限はPrologの持つ制限と変わらない。また、組み込み述語自身の制限についても、Prologと同等か、むしろ算術比較演算子（"<"）などでは本手法の制限の方が緩いといえる。

５－２．本手法の有効性

本手法による実際の仕様の変換を試み、本手法の有効性を検証した。具体例として、図書館の書庫管理システムの仕様を、本言語を用いて定義し、本手法の適用を試みた。仕様の操作定義中の論理式に用いられた述語の数は52個であり、すべての操作定義の論理式について変換可能だった。

仕様中の貸出者登録を行なう操作定義について、実際に実行可能コードの実行時間を測定した。比較のために、この論理式をホーン節形式に変換し、Prolog処理系で動作させた。ただし実行環境は、IBM 3090-200上のVM/370システム・CMSで、VM/Programming in Logic, Lisp/VMの処理系上で実行した。図６に測定結果を示す。この表から、本手法によって得られる実行可能コードが妥当な効率であることがわかり、本手法の有効性が確かめられる。

集合の要素数	実行時間〔ms〕	
	Prolog	Lisp
2000	49	18
5000	123	45
10000	246	86

図６．実行効率の比較

６．おわりに

論理に基づいた仕様記述を実行可能形式へ変換する手法の概略について述べた。現在のところ、本手法の有効性を詳しく検証するために、本手法を用いた変換システムのプロトタイプを作成中である。今後はさらに変換規則の拡充を進める予定である。

参考文献

〔1〕宮本 勲：「ソフトウェア・エンジニアリング：現状と展望」，ＴＢＳ出版会，1982.

〔2〕大野 豊（編）：「新しい時代のソフトウエア」，ｂｉｔ臨時増刊，Vol.16, No.6, 1984.

〔3〕B. Flinn and I.H. Soerensen："CAVIAR：A Case Study in Specification", Programming Research Group, Oxford University Computing Laboratory, June 1985.

〔4〕佐藤 泰介，玉木 久夫：「第一階コンパイラ」，コンピュータソフトウエア，Vol.5, No.2, pp.69-80, 1988.

〔5〕S. Goto："Program Synthesis from Natural Deduction Proofs", Proc. 6th International Joint Conferrence on Artificial Intelligence, pp.339-341, 1979.

〔6〕淵 一博（監修）：「プログラム変換」，共立出版，知識情報処理シリーズ７，1987.

〔7〕R. Kowalski："Logic Programming", Proc. IFIP 83, North Holland Publishing Co., pp.133-145, 1984.

〔8〕長尾 真，淵 一博：「論理と意味」，岩波書店，岩波講座情報科学－７，1983.

〔9〕小野 康一，河野 誠一，門倉 敏夫，深澤 良彰：「論理に基づいた仕様から実行可能形式への変換手法」，情報処理学会第36回全国大会予稿，3M-4, pp.1009-1010, 昭和63年前期.