ソフトウェア基礎論　17－9
（1986. 7. 4）

コンパイラの代数的仕様記述法

An Algebraic Specification Method of Compilers

酒井正彦　北　英彦　坂部俊樹　稲垣康善
Masahiko SAKAI　Hidehiko KITA　Toshiki SAKABE　Yasuyoshi INAGAKI

名古屋大学工学部　三重大学工学部
Nagoya University　Mie University

あらまし　本稿では，コンパイラの代数的仕様記述法を提案し，その記述例としてＰＬ／０コンパイラの仕様記述を与える．本記述法では，コンパイラをソース言語の構文領域からターゲット言語の構文領域への関数とみなし，これを二つの構文領域に対して，コンパイリング関数，補助領域，補助関数を加えて拡張した抽象データ型として等式を用いて記述する．

Abstract　In this paper, we propose an algebraic specification method of compilers, and give a specification of PL/0 compiler as an example. The basic idea of this method is as follows : A compiler is regarded as the function from the syntactic domain of source language to that of target language. The compiler is specified by using equations as an abstract data type which is the two syntactic domains enriched with the compiling function, auxiliary domains and auxiliary functions.

１．　はじめに

プログラミング言語の代数的仕様記述法は，コンパイラ自動生成の理論的枠組みとして有望であり，近年，これを用いたコンパイラの仕様記述法やコンパイラの自動生成に関する研究が盛んに行われ始めている．[1],[2],[3],[4] これらの内で，図１に示す図式に基づくアプローチ[1],[2],[3] では，コンパイラを次の三つの写像

①　ソース言語の意味写像，
②　意味領域間の変換，
⑤　ターゲット言語の意味写像の逆写像

の合成写像としてとらえている．この方法は，仕様中にソース言語とターゲット言語の意味領域の記述が含まれているので，仕様の検証には都合がよいが，仕様の記述量が多くなってしまう．一方，実際にコンパイラを生成する立場からは，コンパイラの仕様記述にソース言語やターゲット言語の意味をも記述するのは負担が大きい．そこで，コンパイラはソースプログラムからターゲットプログラムへの字面から字面への変換システムであり，コンパイラの仕様にはコンパイラの動作（意味）の記述が本質的で，ソース言語やターゲット言語の意味記述は必要ないと考えることにする．すると，コンパイラは図１の中の③，すなわち，ソース言語の構文領域からターゲット言語の構文領域への写像として自然にとらえることができる．しかし，コンパイラの仕様の正当性検証の際には，ソース言語ならびにターゲット言語の意味領域の記述が必要になる．

また，従来の方法では，②の意味領域間の変換を準同型写像として与えようとしているために，ターゲット言語の意味領域の拡張が必要となっている．しかし，この拡張された領域はターゲット言語には関係がなく，むしろ，コンパイラに本質的な領域であり，ターゲット言語の拡張としてとらえるのは不自然である．例えば，文献[5] のＰＡＳＣＡＬコンパイラの代数的仕様記述では，②の意味領域間の変換を準同型写像で与えている．その際，ターゲット言語の意味領域の拡張が不十分なため，メタ記号の導入を余儀なくされている．しかし，これらのメタ

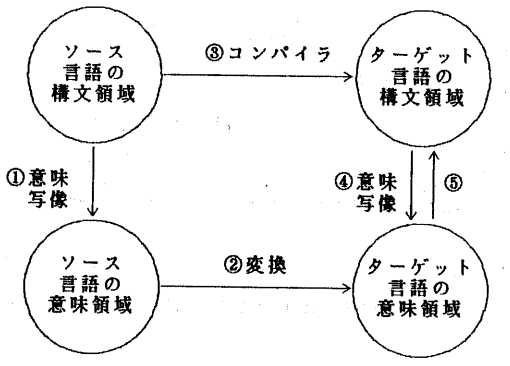

図１　従来の枠組み

記号は，ＬＩＳＰで実現された関数名であり，数学的な意味付けがなされておらず，形式性を損なう原因になっている．

そこで，我々は，(1)コンパイラをソース言語の構文領域とターゲット言語の構文領域の関係としてとらえ，(2)その関係を準同型写像としてではなく，ソース言語とターゲット言語の二つの構文領域の拡張領域上の関数として与えることによって，上に述べた問題を解決したコンパイラの代数的仕様記述法を考案した．また，これを用いて，プログラミング言語ＰＬ／０のコンパイラを仕様記述したのでここに報告する．

２．コンパイラの代数的仕様記述法

本記述法は，コンパイラをソース言語の構文領域からターゲット言語の構文領域への関数とみなし，その関数を等式によって仕様記述する方法である．ただし，二つの構文領域の他に補助の領域や関数を加えることができる．　すなわち，図２に示すように，ソース言語，ターゲット言語の構文領域の他に補助ソート，補助関数，および，コンパイリング関数f_0を加えて拡張した抽象データ型（コンパイリング領域）としてコンパイラをとらえ，その抽象データ型を等式によって記述するものである．

その仕様は，大きく分けて三つの部分からなり，その内の二つはソース言語とターゲット言語の構文領域の仕様であり，もう一つは，それらの二つの領域の拡張であるコンパイリング領域の仕様である．以下では，それらを順に定義する．なお，記法および代数的概念については文献[6],[7]を参照されたい．

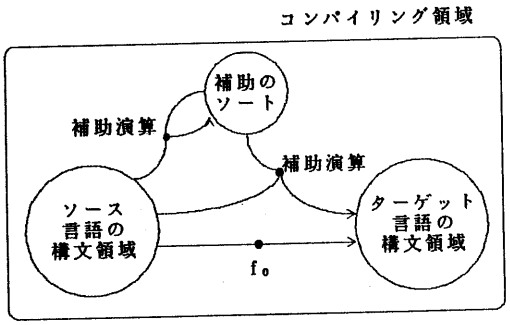

図２　本記述法の枠組み

【定義１（ソース構文仕様）】

ソース言語の構文領域の仕様は，文脈自由文法

$SG=\langle SV_T, SV, SP, SS_0\rangle$

である．ここで，

SV_Tは終端記号の集合，

ＳＶは非終端記号の集合，

ＳＰは生成規則$p: N\to\alpha$の集合で，

ｐは生成規則の名前，$N\in SV$，

$\alpha\in(SV_T\cup SV)^*$，

SS_0はＳＶの要素で開始記号

である．□

文献[8]に述べられているように，文脈自由文法ＳＧをシグニチャ

$SG=\langle SV, SP'\rangle$

とみなすことができる．ここで，

(1)非終端記号の集合ＳＶをソートの集合とする．

(2)生成規則の集合ＳＰから演算記号の集合ＳＰ'を次のように定める．

$SP'=\langle SP'_{w,s}\rangle$，

$w\in SV^*$，$s\in SV$

$SP'_{w,s}=\{p \mid p: N\to\alpha\in SP, s=N, w=nt(\alpha)\}$

である．ただし，ｎｔ（α）は，αからＳＶの要素のみを取り出してできる記号列である．

このようにして，仕様ＳＧが定めるソース言語の構文領域$\mathcal{L}$（ＳＧ）を次のように定義する．

【定義２（ソース構文領域）】

ソース言語の構文領域$\mathcal{L}$（ＳＧ）は，　項代数$\mathcal{T}$［ＳＧ］である．□

直観的には，ソース構文領域は，生成規則の名前を木の節点とし，終端記号が取り除かれた構文木の集合である．

ターゲット言語の構文領域もソース言語と同様に定義する．

【定義３（ターゲット構文仕様）】

ターゲット言語の構文領域の仕様は，文脈自由文法

$TG=\langle TV_T, TV, TP, TS_0\rangle$

である．ここで，

TV_Tは終端記号の集合，

ＴＶは非終端記号の集合，

ＴＰは生成規則$p: N\to\alpha$の集合で，

ｐは生成規則の名前，$N\in TV$，

$\alpha\in(TV_T\cup TV)^*$，

TS_0 はＴＶの要素で開始記号
である．□

【定義４（ターゲット構文領域）】
ターゲット言語の構文領域$\mathcal{L}$（ＴＧ）は，項代数$\mathcal{T}$［ＴＧ］である．□

以上で定義されたソース構文領域およびターゲット構文領域は，コンパイラの入出力を表す抽象データ型である．したがって，コンパイラの仕様はソース構文領域およびターゲット構文領域の拡張であるコンパイリング領域の仕様として与えられる．この拡張は，コンパイラを表す関数であるコンパイリング関数と，その記述に必要となる補助ソート，補助関数の追加によってなされる．

【定義５（コンパイリング領域の仕様）】
コンパイリング領域の仕様$\mathcal{C}$は，６項組
$\mathcal{C}=\langle SG, TG, \Sigma, \mathcal{V}, \mathcal{A}, f_0\rangle$
である．ここで，
ＳＧはソース構文仕様，ＴＧはターゲット構文仕様で，$SV \cap TV = \phi$，
Σはシグニチャ$\langle \mathcal{S}, \mathcal{F}\rangle$で，
$SV \cup TV \subseteq \mathcal{S}$，$SP' \cup TP' \subseteq \mathcal{F}$，
$\mathcal{V}$は変数集合族，$\mathcal{A}$は等式$\xi = \eta$の集合，
$f_0 \in \mathcal{F}_{SS_0, TS_0}$はコンパイリング関数記号
である．□

コンパイリング領域は，その仕様$\mathcal{C}$から定められ，次に示す抽象データ型となる．

【定義６（コンパイリング領域）】
コンパイリング領域の仕様$\mathcal{C}$から定まるコンパイリング領域ＳＤ（$\mathcal{C}$）は仕様$\mathcal{C}$が，ソース構文領域$\mathcal{L}$（ＳＧ），ターゲット構文領域$\mathcal{L}$（ＴＧ）に対して，完全性ならびに無矛盾性[6],[7]を満たすとき，商代数$\mathcal{T}$［Σ］／≡である．ここで，≡は$\mathcal{S}$の要素ｓに対して$\langle \equiv_s \rangle$であり，次のように定められる項代数$\mathcal{T}$［Σ］上の合同関係である．
$\xi, \eta \in \mathcal{T}$［$\Sigma$］について，
$\xi \equiv_s \eta$　ｉｆｆ　$\mathcal{A} \vdash \xi \approx \eta$　□

直観的には，コンパイリング関数は，仕様記述されたコンパイラそのものを特徴づけている．すなわち，ソースプログラムをコンパイルすることは，ソースプログラムの構文木ξおよびコンパイリング関数記号f_0に対して次式
$f_0(\xi) \equiv_s \eta$
を満たすターゲットプログラムの構文木ηをみつけることである．

３　ＰＬ／０コンパイラの仕様記述

本記述法の有効性を示すために，ＰＬ／０コンパイラ[9]を仕様記述した．ＰＬ／０コンパイラは，Ｗｉｒｔｈがコンパイラ作成の一例として示したものであり，コンパイラ自動生成の研究では既におなじみのものである．

ソース言語であるＰＬ／０は，簡単な言語であるが，言語としての基本的な制御機能（条件文，繰り返し文，手続き呼び出し文など）をもっている．この言語の構文領域の仕様を図３に与える．これは，文献[9]の構文図を生成規則の形に直しただけのものである．

前節で述べたように，各々の生成規則はソース言語の構文領域の演算記号と考えられる．　例えば，ｗｈｉｌｅ文の生成規則

```
STM_WHILE
    : stm →  'while' cond 'do' stm
```

は，ｃｏｎｄとｓｔｍを引数に持ち，ｓｔｍを返す演算記号ＳＴＭ＿ＷＨＩＬＥとなる．

次に，ターゲット言語の構文領域の仕様を図４に示す．この言語はＰＬ／０計算機と呼ばれるスタック・マシンの命令系列である．命令には，ｌｉｔ（数値のロード），ｌｏｄ（変数のロード），ｓｔｏ（変数へのストア），ｃａｌ（サブルーチンの呼び出し），ｉｎｔ（変数領域の確保），ｊｍｐ（無条件分岐），ｊｐｃ（零判定分岐），ｏｐｒ（算術演算）の７種類がある．

コンパイリング領域の仕様は，コンパイラの意味であるコンパイリング関数とその記述に必要となる補助関数を持つ抽象データ型の記述である．ここでは，コンパイリング関数記号f_0としてＣＯＭＰを

```
SG = <SV_T , SV, SP, SS_0 >
SV_T = {begin, end, if, then, ... }
SV = {program, block, stm, stm_list,
      expr, cond, ident        ... }
SP = {
   PROG  : prog  → block '.'
   BLOCK : block → const_part var_part
                   proc_part stm
   STM_WHILE
         : stm →  'while' cond 'do' stm
   STM_ASSIGN
         : stm →  ident ':=' expr
   ... }
SS_0 = prog
```

図３　ソース構文領域の仕様

用いる．ＣＯＭＰは次の等式で定義されるソース言語の構文木からターゲット言語の構文木への関数である．

```
COMP(p)
== COMP_PROG(p,INIT_C_SYMTAB(),ZERO())
```

ここで，ＩＮＩＴ＿Ｃ＿ＳＹＭＴＡＢは空の記号表を返す演算，ＣＯＭＰ＿ＰＲＯＧは第一引数のプログラムを第二引数の記号表を用いて第三引数の番地で始まる命令系列にコンパイルする演算である．さらに，この演算ＣＯＭＰ＿ＰＲＯＧも他の補助演算で定義されている．

以下では，繰り返し文のコンパイルに注目しよう．繰り返し文に対応するＰＬ／０構文木の部分木は，ＳＴＭ＿ＷＨＩＬＥ（ｃ，ｓ）の形をしている．これをコンパイルする演算はＣＯＭＰ＿ＳＴＭであり，次の等式によって定義される．

```
COMP_STM(STM_WHILE(c,s),t,a)
== INST_L2(
     APPEND_INST(
       INST_L2(
         COMP_COND(c,t,a),
         INST_JPC(
           NEXT_ADR_COND(c,a),
           NEXT_ADR_STM(
             STM_WHILE(c,s),a))),
       COMP_STM(
         s,t,
         ADD1_INT(NEXT_ADR_COND(c,a)))),
     INST_JMP(
       SUB1_INT(
         NEXT_ADR_STM(STM_WHILE(c,s),a))))
```

ここで，ＩＮＳＴ＿Ｌ２はターゲット言語の構文木を構成する演算記号で，第一引数に命令系列を，第二引数に命令をとる．　ＡＰＰＥＮＤ＿ＩＮＳＴは二つの命令系列をつないで一つの命令系列にする補助演算，ＣＯＭＰ＿ＣＯＮＤは条件部をコンパイルする演算，　ＡＤＤ１＿ＩＮＴは１を加える演算，

$TG = < TV_T, TV, TP, TS_0 >$

TV_T = {lod, sto, cal, jmp, jpc, ... }

TV = {t-prog, inst, inst_l, num }

TP = {

```
  T-PROG   : t-prog → inst_l
  INST_L   : inst_l →
  INST_L2  : inst_l → inst_l inst
  INST_JMP : inst   → int 'jmp' '0' ',' int
  INST_JPC : inst   → int 'jpc' '0' ',' int
  INST_LOD : inst   → int 'lod' int ',' int
  INST_STO : inst   → int 'sto' int ',' int
  ... }
```

図４　ターゲット構文領域の仕様

ＳＵＢ１＿ＩＮＴは１を引く演算である．　また，ＮＥＸＴ＿ＡＤＲ＿ＳＴＭ，ＮＥＸＴ＿ＡＤＲ＿ＣＯＮＤはそれぞれ第一引数の構文木を第二引数の番地からコンパイルしたときのアドレス・カウンタの値を返す演算である．コンパイリング領域の仕様を

$C = < SG, TG, \Sigma, \mathcal{V}, \mathcal{A}, f_0 >$

$\Sigma = < S, \mathcal{F} >$

S = SVUTVU

　{c_symtab, info, int, id, bool }

$\mathcal{F}$ = SP'UTP'U {

```
  COMP       : prog → t-prog
  COMP_PROG  : prog,c_symtab,int  →  t-prog
  COMP_BLOCK : block,c_symtab,int →  inst_l
  COMP_STM   : stm,c_symtab,int   →  inst_l
  COMP_EXPR  : expr,c_symtab,int  →  inst_l
  COMP_COND  : cond,c_symtab,int  →  inst_l
  NEXT_ADR_STM : stm,int          →  int
  NEXT_ADR_EXPR: cond,int         →  int
  NEXT_ADR_COND: expr,int         →  int
  ... }
```

$\mathcal{V}$ = {p:prog, b:block, s:stm, e:expr,
　t:c_symtab, x:id, a:int　　... }

$\mathcal{A}$ = {

```
  COMP(p)
  == COMP_PROGRAM(p,INIT_C_SYMTAB(),ZERO())

  COMP_PROG(PROG(b),t,a)
  == T-PROG(COMP_BLOCK(b,t,a))

  COMP_STM(STM_WHILE(c,s),t,a)
  == INST_L2(
       APPEND_INST(
         INST_L2(
           COMP_COND(c,t,a),
           INST_JPC(
             NEXT_ADR_COND(c,a),
             NEXT_ADR_STM(
               STM_WHILE(c,s),a))),
         COMP_STM(
           s,t,
           ADD1_INT(NEXT_ADR_COND(c,a)))),
       INST_JMP(
         SUB1_INT(
           NEXT_ADR_STM(STM_WHILE(c,s),a))))

  NEXT_ADR_STM(STM_WHILE(c,s),a)
  == ADD_INT(
       TWO(),
       NEXT_ADR_STM(s,NEXT_ADR_COND(c,a)))

  COMP_STM(STM_ASSIGN(x,e),t,a)
  == INST_L2(
       COMP_EXPR(e,t,a),
       INST_STO(
         NEXT_ADR_EXPR(e,a),
         C_RET_BLOCK(x,t),
         MK_INFO-INT(C_RET_SYM(x,t))))

  NEXT_ADR_STM(STM_ASSIGN(x,e),a)
  == ADD1_INT(NEXT_ADR_EXPR(e,a))

  ... }
```

f_0 =　COMP

図５　コンパイリング領域の仕様

図５に示す．なお，仕様記述には，１１１個の演算記号と１０２個の公理を用いた．ただし，ｂｏｏｌ型，整数型，識別子型に関するものはこれらの数に含まれていない．

4　コンパイラ生成系の実現

本記述法に基づくコンパイラ生成系は，以下に示すように比較的容易に実現できる．

(1)　ソースプログラムの構文解析を行って，ソース構文領域上の項を出力するフェーズの生成には，Ｙａｃｃ[10]など，構文の仕様から構文解析プログラムを生成するツールがそのまま利用できる．

(2)　(1)で得られた項にコンパイリング関数f_0を適用して，ターゲット構文領域の項を計算するフェーズの生成は，コンパイリング領域を実現することにほかならない．これには，Ｄｉｍｐｌｅ[11]などの抽象データ型直接実現システムを利用することができる．

(3)　(2)で得られた項をターゲットプログラムに変換するフェーズは，構文木が表す文字列の生成，すなわち，生成規則による文字列の導出を行う．この作業は，構文解析の逆変換であり，構文解析よりもはるかに易しい．

我々は，上で述べた三つのフェーズの生成に，各々，Ｙａｃｃ，Ｄｉｍｐｌｅ，Ｙａｃｃを利用して，３節で説明したＰＬ／０コンパイラの仕様から実際にＰＬ／０コンパイラを生成した．なお，仕様から各々のツールの入力への書き換えは人手で行ったが，具体的な仕様記述言語を定めさえすれば，自動変換プログラムの実現は容易である．

ここで，　仕様から生成されたＰＬ／０コンパイラの動作例を示す．　図６は，最大公約数を求めるＰＬ／０プログラムである．このプログラムのソース構文領域上の項は図７のようになる．これに，コンパイリング関数ＣＯＭＰを適用すると図８に示すターゲット構文領域上の項が得られる．最後に，こ

```
const m=84, n=36;
var   x,y,z;

procedure GCD;
  var f,g;
  begin
    f:=x; g:=y;
    while f<>g do
      begin
        if f<g then g:=g-f;
        if g<f then f:=f-g
      end;
    z:=f
  end;

begin
  x:=m; y:=n; call GCD;
end.
```

図６　ＰＬ／０プログラム

```
(prog
 (block
  (const_part
   (const_l2 (const_l (const_def 'm '84))
    (const_def 'n '36)))
  (var_part
   (var_l2
    (var_l2 (var_l (var_name 'x)) (var_name 'y))
    (var_name 'z)))
  (proc_part
   (proc_l
    (proc_dcl 'gcd
     (block (const_part2)
      (var_part
       (var_l2
        (var_l (var_name 'f)) (var_name 'g)))
      (proc_part2)
      (stm_block
       (stm_l2
        (stm_l2
         (stm_l2
          (stm_l (stm_assign 'f (expr_sym 'x)))
          (stm_assign 'g (expr_sym 'y)))
         (stm_while
          (cond_ne (expr_sym 'f) (expr_sym 'g))
          (stm_block
           (stm_l2
            (stm_l
             (stm_if
              (cond_lt (expr_sym 'f) (expr_sym 'g))
              (stm_assign 'g
               (expr_diff
                (expr_sym 'g) (expr_sym 'f)))))
             (stm_if
              (cond_lt (expr_sym 'g) (expr_sym 'f))
              (stm_assign 'f
               (expr_diff
                (expr_sym 'f) (expr_sym 'g))))))))))
        (stm_assign 'z (expr_sym 'f))))))))
  (stm_block
   (stm_l2
    (stm_l2
     (stm_l2 (stm_l (stm_assign 'x (expr_sym 'm)))
      (stm_assign 'y (expr_sym 'n)))
     (stm_call 'gcd))
    (stm_null)))))
```

図７　ＰＬ／０プログラムの項

```
(t-program (inst_list2 (inst_list2 (inst_list2
(inst_list2 (inst_list2 (inst_list2 (inst_list2
(inst_list2 (inst_list2 (inst_list2 (inst_list2
(inst_list2 (inst_list2 (inst_list2 (inst_list2
(inst_list2 (inst_list2 (inst_list2 (inst_list2
(inst_list2 (inst_list2 (inst_list2 (inst_list2
(inst_list2 (inst_list2 (inst_list2 (inst_list2
(inst_list2 (inst_list2 (inst_list2 (inst_list2
(inst_list2 (inst_list2 (inst_list2 (inst_list2
(inst_list2 (inst_list2 (inst_list2 (inst_list)
(inst_jmp 0 31))   (inst_jmp 1 2))
(inst_int 2 5))   (inst_lod 3 1 3))
(inst_sto 4 0 3))   (inst_lod 5 1 4))
(inst_sto 6 0 4))   (inst_lod 7 0 3))
(inst_lod 8 0 4))   (inst_ne 9))
(inst_jpc 10 28))   (inst_lod 11 0 3))
(inst_lod 12 0 4))   (inst_lt 13))
(inst_jpc 14 19))   (inst_lod 15 0 4))
(inst_lod 16 0 3))   (inst_diff 17))
(inst_sto 18 0 4))   (inst_lod 19 0 4))
(inst_lod 20 0 3))   (inst_lt 21))
(inst_jpc 22 27))   (inst_lod 23 0 3))
(inst_lod 24 0 4))   (inst_diff 25))
(inst_sto 26 0 3))   (inst_jmp 27 7))
(inst_lod 28 0 3))   (inst_sto 29 1 5))
(inst_ret 30))   (inst_int 31 6))
(inst_lit 32 84))   (inst_sto 33 0 3))
(inst_lit 34 36))   (inst_sto 35 0 4))
(inst_cal 36 0 1))   (inst_ret 37)))
```

図８　ＰＬ／０計算機のプログラムの項

の項をターゲットプログラムに直すと図９のようになる．

```
0 jmp 0,31        19 lod 0,4
1 jmp 0,2         20 lod 0,3
2 int 0,5         21 opr 0,10
3 lod 1,3         22 jpc 0,27
4 sto 0,3         23 lod 0,3
5 lod 1,4         24 lod 0,4
6 sto 0,4         25 opr 0,3
7 lod 0,3         26 sto 0,3
8 lod 0,4         27 jmp 0,7
9 opr 0,9         28 lod 0,3
10 jpc 0,28       29 sto 1,5
11 lod 0,3        30 opr 0,0
12 lod 0,4        31 int 0,6
13 opr 0,10       32 lit 0,84
14 jpc 0,19       33 sto 0,3
15 lod 0,4        34 lit 0,36
16 lod 0,3        35 sto 0,4
17 opr 0,3        36 cal 0,1
18 sto 0,4        37 opr 0,0
```

図９　ＰＬ／０計算機のプログラム

５．まとめ

本報告では，コンパイリングをソース言語の構文領域からターゲットの構文領域への関数とみなして，自然かつ単純な枠組みを用いたコンパイラの代数的仕様記述法を考案し，ＰＬ／０コンパイラの仕様記述を与えた．

さて，本方法における仕様の正当性は，従来から用いられている図式（図１）に基づいて定式化できる．すなわち，①②の合成関数と③④の合成関数が等しいとき，①は②③④に関して正当であると定義する．我々は，ＰＬ／０サブセットのコンパイラに対して，既に，この証明を行っている．本報告で示したＰＬ／０コンパイラの正当性の証明は今後の課題であり，これらについては，次の機会に報告する．

＜謝辞＞

日頃御指導賜る豊橋技術科学大学本多波雄学長，名古屋大学福村晃夫教授，並びに御討論下さる阿曽弘具助教授をはじめ研究室の皆様に感謝致します．

なお本研究は，一部，文部省科研費（一般(c) 課題番号60550263，特定研究(1)多元知識情報課題番号61102003）および，倉田奨励金の援助による．

＜文献＞

(1) Gaudel M. C. : Specification of Compilers as Abstract Data Types Representations, Proc. Workshop on Semantics-Directes Compilers in Aarhus, LNCS94, pp140-164, (1980)

(2) Deschamp P. : PERLUETTE : A Compiler Producing System using Abstract Data Types, Proc. 5th Coll in Programming, LNCS137, pp63-77, (1982)

(3) Thatcher J. W., Wagner E. G., Wright J. B. : More on Advice on Structuring Compilers and Proving them Correct, Proc. Workshop on Semantics-Directed Compilers in Aarhus, LNCS94, pp165-188, (1980)

(4) Mosses P. : A Constructive Approach to Compiler Correctness, Proc. Workshop on Semantics-Directed Compilers in Aarhus, LNCS94, pp165-188, (1980)

(5) Despeyroux J. : An Algebraic Specification of PASCAL Compiler, INRIA, Rapports de Recherch, 210, (1983)

(6) 稲垣，坂部：抽象データタイプの代数的仕様記述の基礎(1) －多ソート代数と等式論理，情報処理，Vol.25, No.1, pp47-53, (1984,1)

(7) 稲垣，坂部：抽象データタイプの代数的仕様記述の基礎(2) －抽象データタイプ，情報処理，Vol.25, No.5, pp491-501, (1984,1)

(8) Thatcher J. W., Wagner E. G. Wright J.B. : Initial Algebra Semantics and Continuous Algebra, JACM, Vol.24, No.1, pp68-95, (1977)

(9) Wirth N. : Algorithms + Data Structure = Programs, Prentice-hall, (1976)

(10)Johnson S. C. : Yacc - Yet Another Compiler-Compiler, Comp. Sci. Tech. Rep., No.39, Bell-Lab., (1975)

(11)川辺，坂部，稲垣，本多：抽象データ型の直接実現システム，電子通信学会，技術報告AL83-65, (1984)