ソフトウエア工学 93－7
（1993．7．8）

部分計算を用いた等式仕様の変換と実行

森 彰* 篠原 伸生* 前田 将徳** 松本 吉弘*
京都大学工学部* 松下電器**

本稿では、ソフトウェアの等式仕様を、項書換え系インタプリタとともに部分計算することにより、手続き型言語に変換する一つの方法について述べる。もとの仕様には疑似関数を加えておき、部分計算したのちにこれを手続き型言語による実関数で置き換える。これにより、プログラムの重要な部分は等式仕様で、入出力やアルゴリズム的記述などの操作的仕様は手続き型言語でといった、等式仕様の形式性と手続き型言語の利便性を融合したプロトタイピングが可能になる。簡単な行エディタを例に取って、C言語で直接プログラム作成した場合との比較を行い、本手法の評価を行う。

Transformation and Execution of Equational Specifications Using Partial Evaluation.

Akira MORI* Nobuo SHINOHARA* Katsunori MAEDA** Yoshihiro MATSUMOTO*
Faculty of Engineering, Kyoto University*
Yoshida honmachi, Sakyo-ku, Kyoto 606-01, JAPAN
Matsushita Electric Industrial Co., LTD.**
1006, Kadoma, Kadoma-Shi, Osaka 571, JAPAN

In this paper, we propose a transformation method for equational specifications using partial evaluation techniques. The specification is transformed into imperative programs through partial evaluation with an interpreter for term rewriting systems. By replaing *psuedo functions* attached to the specification with *real functions* written in imperative language, we obtain a prototyping method utilizing both of the rigor of equational specifications and the convenience of imperative languages. We take a simple line editor as a running example and make comparison with the direct coding in the C language.

1 はじめに

ソフトウェア開発においては、開発初期に与えられる形式的仕様を段階的に詳細化 (refinement) していき、最終的に実行可能なプログラムへと至るのが望ましい。代数的仕様は、形式的仕様を記述するための方法であり、ソフトウェアの仕様を多ソート代数に基づいた等式論理で記述することにより、始代数意味論などの表示的意味を厳密に定義することができる。

一方、等式による仕様記述すなわち等式仕様は、項書換え系として各等式に「左辺を右辺に書き換える」という意味を与え、仕様に操作的意味を持たすことができる。項書換え系は数学的な解析が可能な計算モデルであるので、仕様の変換・検証といった技術を、厳密に議論することが可能である。この点で等式仕様は、ソフトウェア開発、特にプロトタイピングにおいて有用な方法であると言える [1, 2]。

反面、等式仕様は本来静的な意味記述であるため、たとえ項書換え系として実行するにしても、順序依存や局所状態を持つアルゴリズムのような動的な操作は表現しにくい。例えば、キー入力、画面表示、ファイル操作、文字列検索などは、等式仕様で記述・実行することは困難である。

本研究ではこの問題点に対する一つの解法として、項書換え系インタプリタを等式仕様と共に部分計算し、同等の意味を持つ手続き型のプログラムに変換した後に、必要な拡張を加え実行可能プログラムとする手法を提案する。すなわち、プログラムの意味的に重要な骨組は等式仕様で記述し、入出力などあまり重要でない部分は手続き型言語で記述する。この方法を用いれば、プログラムの厳密な検証などは等式仕様の利点をそのまま受け継ぐことができ、形式的仕様を用いたソフトウェアのプロトタイピングに役立つものと思われる。

以下、2章では部分計算について、3章では等式仕様と項書換え系インタプリタについてそれぞれ説明し、4章では例として簡単な行エディタの等式仕様を取り上げ、その変換と実行について説明する。そして5章でC言語で直接行エディタを作成した例との比較を行ない、6章で考察を加える。

```
f(n,x) =  if n=0 then 1
          else if even(n) then f(n/2,x)^2
          else x*f(n-1,x)

g(x)   =  f(5,x)
       =  x*(x^2)^2
```

図 1: 部分計算の例

2 部分計算

この章では、部分計算の概念、本研究に使用した部分計算器 *Similix*[5] について述べる。

2.1 部分計算とは

プログラムの部分計算とは、「稼働環境に関する情報を利用して、汎用のプログラムをより能率のよいプログラムに特化すること」[3] である。あるプログラムといくつかの既知の入力データが与えられた時、これらを部分計算して得られるプログラムを剰余プログラム (residual programs) と呼ぶ。この際、残りの入力データを剰余プログラムに代入した実行結果は、元のプログラムに全入力データを代入した実行結果と全く同じでなければならない。

例えば、図 1のようなプログラム (関数) $f(n,x)$ があるとする。これを n=5 で部分計算すると、剰余プログラム $g(x)$ が生成される。$f(5,8)$ と $g(8)$ を計算すると同じ値が得られるが、計算速度は $g(8)$ のほうが速いことは明らかであろう。部分計算器を PE で表すことにすると、上の部分計算は $PE(f(n,x), 5) = g(x)$ で表せる。但し、PE の第一引数はプログラム、第二引数は既知の変数の値、右辺は剰余プログラムである。

また、二村により次のような部分計算についての性質が発見されている。すなわち、int をあるプログラム言語のインタプリタ、そして p をプログラムとする時、

- PE(int,p)= 目的プログラム (第一射影)
- PE(PE,int)= コンパイラ (第二射影)
- PE(PE,PE)= コンパイラ・コンパイラ (第三射影)

```
(define (append1 l1 l2)
  (if (null? l1)
      l2
      (cons (car l1) (append1 (cdr l1) l2))))
```

図 2: 関数 append1

となることが知られている [3, 4]。ただし、第二、第三射影のためには PE は自己適用可能でなければならない。本研究では第一射影の性質を利用し、項書換え系インタプリタを書き換え規則と共に部分計算し、その書き換え規則に特化されたインタプリタを目的プログラムとして生成する。

2.2 部分計算器 *Similix*

Similix[5] はコペンハーゲン大学で開発された自己適用可能な部分計算器で、プログラム言語 Scheme 上で動作し、分岐、大域変数などを除いたサブセットについてユーザの注釈などなしに自動的に部分計算を行なう。

Similix は、次の四つのフェーズに分けて部分計算を行なっている。

1. ***Front-end***: 部分計算する目的プログラム P を *Similix* の中で扱いやすい抽象的な文法で書かれたプログラム P_1 に変換する。
2. ***Preprocessor***: P_1 の（再帰呼び出しも含めた）全ての変数について未知か既知かを調べ、それに基づいて各関数が展開可能か不可能かを決定する。その結果を各関数に注釈として付け、それを P_2 として次に渡す。
3. ***Specializer***: P_2 に実際の既知の入力値を与え、注釈通り展開できる関数は展開する。それを P_3 として次に渡す。
4. ***Postprocessor***: *Specializer* で展開した関数を更に展開してプログラムを整理し、剰余プログラム R として出力する。

Similix の動作例を図 3にあげる。二つのリストをつなげる関数 append1(図 2) の第一引数をリスト (1 2 3) に固定して部分計算を行なったものである。

```
1:  ==> (load "append.sim")
    ;Loading  append.sim into SCHEME-ENV
    APPEND1

2:  ==> (append1 '(1 2 3) '(4 5 6))
    (1 2 3 4 5 6)

3:  ==> (similix 'append1 (list '(1 2 3) '***) "append.sim")
    front-ending cl bt sp eod oc rl
    specializing
    ((LOADT (STRING-APPEND **SIMILIX-LIBRARY** "scheme.adt"))
    (DEFINE (APPEND1-0 L2_0) (CONS 1 (CONS 2 (CONS 3 L2_0)))))

4:  ==> (load-residual-program)
    ()

5:  ==> (append1-0 '(4 5 6))
    (1 2 3 4 5 6)
```

図 3: *Similix* の動作例

3 項書換え系インタプリタ

本研究で作成した部分計算のための項書換え系インタプリタについて説明する。

3.1 項書換え系

項書換え系 [6, 7, 1] は、書き換え規則と呼ばれる向きづけられた等式の集合として定義され、その計算は左辺から右辺への書き換えを繰り返し適用していくことである。ある項書換えが、合流性と停止性を満たす (これを完備な項書換え系と呼ぶ) とき、任意の項に対する書き換えは必ず一意な結果をもって終了することが知られている (この一意な結果を正規形と呼ぶ)。本研究では与えられた等式仕様がすでに完備な項書換え系としてみなせるものとし、前もって別のツールにより Knuth-Bendix のアルゴリズムによる完備化を行なっている。

ここで簡単な例を用いて等式仕様と項書換え系を説明する。図 4に自然数の加算と乗算を定義した等式仕様を示す。これを項書換え系とみなし項 plus(mult(s(0),s(0)),0) を書き換えると、図 5のようになり正規形 s(0) が得られる。下線は書き換えられる部分を示し、図の右の数字は適用した等式を示す。

3.2 項書換え系インタプリタ

本研究で作成した項書換え系インタプリタは、基本的には与えられた項に対して書き換え規則を繰り

```
Eqns = {
   r1 : plus(X,0) → X
   r2 : plus(X,s(Y)) → s(plus(X,Y))
   r3 : mult(X,0) → 0
   r4 : mult(X,s(Y)) → plus(mult(X,Y),X)
        }
```

図 4: 自然数の加算乗算を表す等式仕様

```
   plus(mult(s(0),s(0)),0)
→ plus(plus(mult(s(0),0),s(0)),0)      :r4
→ plus(plus(0,s(0)),0)                 :r3
→ plus(s(plus(0,0)),0)                 :r2
→ plus(s(0),0)                         :r1
→ s(0)                                 :r1
```

図 5: 項 plus(mult(s(0),s(0)),0) の正規形

返し適用していき、その正規系を求めるものである。ただし、*Similix* で部分計算できるように大域変数や分岐を用いないサブセットの Scheme で書かれており、また後述の拡張操作に都合が良いコードを生成するよう工夫がなされている。後者の点は、部分計算をプログラム変換として用いる場合に特に重要な点である。主な特徴は以下の通りである。

- 項は S-式で、書き換え規則は項のリスト、項書換え系は書き換え規則のリストで表現する。
- 書き換え規則の適用は最左最内戦略を用いる。
- 書き換え規則適用のための単一化アルゴリズムは、*Similix* の部分計算アルゴリズムをよく考慮して実現する。

4 等式仕様の変換と実行

本章では、2 章で述べた部分計算の概要と、3 章で述べた項書換え系インタプリタの概要を踏まえて、行エディタの等式仕様を部分計算を用いて変換・実行する過程について報告する。

等式仕様は等式の集合からなり、項書換え系として左辺から右辺への書き換え規則の集合として扱うことができる。ある等式仕様とその上での項を項書換え系インタプリタに入力すると正規形が得られる。しかし、ここでの書き換えはあくまで記号上の書き換えであって、入出力操作、例外処理や込み入ったアルゴリズムを実現するには、時として必要以上の困難さを伴う。また、とりあえず完成した部分的な仕様をもとに実行可能なプログラムを得たい場合も多いと考えられる。

この点に関し本研究では、等式仕様では表現しにくい操作を、関数名だけが存在する仮の関数 (疑似関数) としてプログラムの中に記述しておく。そして、この等式仕様を項書換え系インタプリタに対して部分計算し、もとの等式仕様の項書換えによる実行と同等な Scheme のプログラムを得る。これは、インタプリタ中の単一化のループが各書き換え規則に関して展開されたようなものであって、事前に挿入しておいた疑似関数はそのままの形でプログラムのに残っている。これを Scheme で記述した実際の関数 (実関数) で置き換える。この置き換えは、疑似関数、実関数の対応と、項書換え系インタプリタの構造から決定される剰余プログラムの構造を知っていれば自動的に行なうことができる。こうすることで、等式仕様では記述が困難な操作を、等式仕様と組み合わせて実現することが出来る。この変換と実行の過程を表す概念図を図 6に示す。ここでの例は、図 4での自然数の加乗算の等式仕様に入出力操作を付加するものである。

4.1 行エディタの等式仕様を用いた実行例

ここでは、簡単な行エディタの等式仕様を部分計算を用いて変換し、入出力やファイル操作、文字列検索などの機能を付加する過程について説明する。

4.1.1 行エディタの等式仕様

行エディタとは、文書を行単位で編集するエディタであり、代表的なものに UNIX の *ed* や *ex* などがある。これらは、*emacs* などのスクリーンエディタのように、文字単位で編集することはできない。
図 7に本研究で例として取り上げた簡単な行エディタのための項書換え系を示す (文献 [8] 参照)。

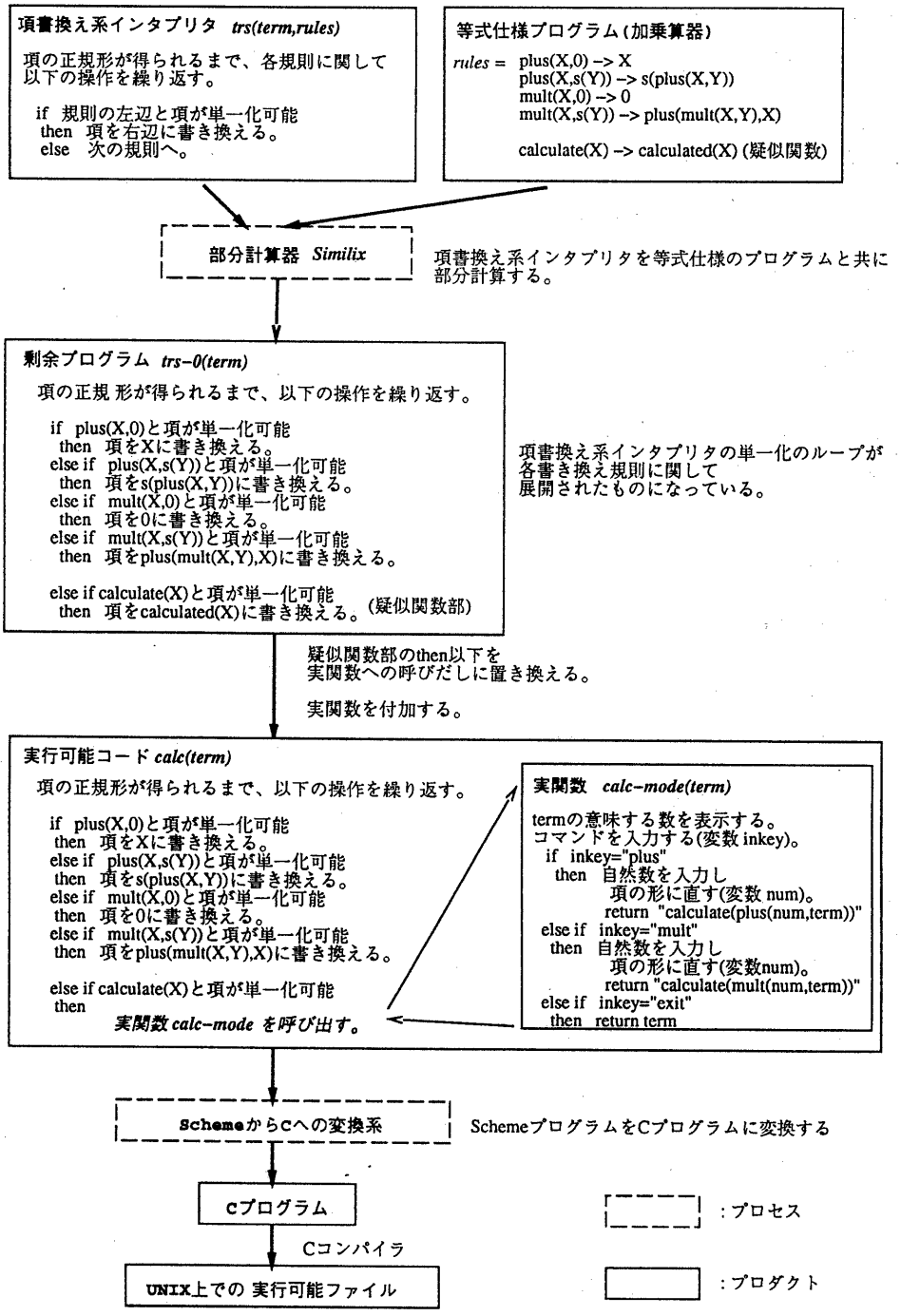

図 6: 部分計算による等式仕様の変換と実行の概念図

```
Eqns={
append(S,page(L1,L2,B)) = page(d-add(L1,S),L2,B)
insert(S,page(empty,L,B)) = page(empty,L,B)
insert(S1,page(d-add(L1,S2),L2,B))
            = page(d-add(d-add(L1,S1),S2),L2,B)
change(S,page(empty,L,B)) = page(empty,L,B)
change(S1,page(d-add(L1,S2),L2,B))
            = page(d-add(L1,S1),L2,S2)
delete(page(empty,L,B)) = page(empty,L,B)
delete(page(d-add(L1,S),L2,B)) = page(L1,L2,S)
yank(page(L1,L2,B)) = page(d-add(L1,B),L2,B)
memory(page(empty,L,B)) = page(empty,L,B)
memory(page(d-add(L1,S),L2,B)) = page(d-add(L1,S),L2,S)
move-up(page(empty,L,B)) = page(empty,L,B)
move-up(page(d-add(L1,S),L2,B)) = page(L1,u-add(S,L2),B)
move-down(page(L,empty,B)) = page(L,empty,B)
move-down(page(L1,u-add(S,L2),B)) = page(d-add(L1,S),L2,B)
top(page(empty,L,B)) = page(empty,L,B)
top(page(d-add(L1,S),L2,B)) = top(page(L1,u-add(S,L2),B))
bottom(page(L,empty,B)) = page(L,empty,B)
bottom(page(L1,u-add(S,L2),B))
            = bottom(page(d-add(L1,S),L2,B))
clear(page(L1,L2,B)) = page(empty,empty,B)
create-page = page(empty,empty,"")
replace(S1,S2,page(L1,L2,B))
            = page(h-repl(S1,S2,L1),l-repl(S1,S2,L2),B)
h-repl(S1,S2,empty) = empty
h-repl(S1,S2,d-add(L,S1)) = d-add(h-repl(S1,S2,L),S2)
h-repl(S1,S2,d-add(L,S3)) = d-add(h-repl(S1,S2,L),S3)
l-repl(S1,S2,empty) = empty
l-repl(S1,S2,u-add(S1,L)) = u-add(S2,l-repl(S1,S2,L))
l-repl(S1,S2,u-add(S3,L)) = u-add(S3,l-repl(S1,S2,L))

## pseudo functions ##

edit(P) = edited(P)
print(P) = printed(P)
buffer(P) = buffered(P)
load(F) = loaded(F)
save(F,P) = saved(F,P)
repl-word(W1,W2,S) = repl-worded(W1,W2,S)
      }
```

図 7: 簡単な行エディタの等式仕様 (一部)

行エディタで扱う項の構造について述べる。行エディタで編集する文書をページと呼ぶことにする。ページは一つのカーソルと上部ページと下部ページから構成され、カーソルが指す行をカレントラインと呼ぶ。上部ページはページのカレントラインより上の行の集合（カレントラインを含む）を、下部ページそれより下の行の集合（カレントラインを含まない）をそれぞれ表す。また、行の削除や変更の際の変更前の行を保存するミニバッファも設け、必要な時はこれを再び挿入することができる。ページは *page*(上部ページ, 下部ページ, ミニバッファ) の書式で表し、上部ページは *d-add* を、下部ページは *u-add* を用いて、ミニバッファは Scheme のストリングの書式でそれぞれ記述する。ページが n 行からなる時（n は非負整数）、カーソルは 0 から n 行目を指すことができる。詳しくは以下を参照されたい。

```
|abcdefg
+hijklmn   <- current line
|opqrstu
|vwxyz

page(d-add(d-add(empty,"abcdefg"),"hijklmn")
     ,u-add("opqrstu",u-add("vwxyz",empty))
     ,"")
```

図 8: ページとその項表現の例

```
     . . . . . . . . . . . . . . . .


(LET ((ENVIRONMENT_41
       (UNIFY-0-4 TERM_1 (QUOTE (EDIT (? P))) ()))
      (IF (NEQ? ENVIRONMENT_41 (QUOTE FAILED))
          (BINDING-0-6 (QUOTE (EDITED (? P)))
                       ENVIRONMENT_41)

(LET ((ENVIRONMENT_42
       (UNIFY-0-4 TERM_1 (QUOTE (PRINT (? P))) ()))
      (IF (NEQ? ENVIRONMENT_42 (QUOTE FAILED))
          (BINDING-0-6 (QUOTE (PRINTED (? P)))
                       ENVIRONMENT_42)

     . . . . . . . . . . . . . . . .
```

図 9: 単一化のループが各書き換え規則について展開された様子 (疑似関数の一部)

具体的なページのイメージとその項表現を図 8に示す。これは典型的なページの例で、+ はカレントラインを示しており 2 行目を指している。前述の様に、カレントラインは 0 行目から 4 行目を指すことが出来る。なお、今回の例ではミニバッファは空である。

4.1.2 変換の過程

まず、項書換え系インタプリタを行エディタの等式仕様より得られる項書換え系に関して部分計算する。そして、部分計算後の剰余プログラム中の疑似関数の書き換え規則に対する単一化の `if` 文について、`then` 以下を実関数への関数呼びだし文に書き換える (図 9)。

次に実関数を Scheme のプログラムとして作成する。実関数に対する入力と出力は、剰余プログラムの構造により決定される。すなわち、ページ全体の表現を受けとって、適当な操作を加えた後のページ全体の表現を返すというパターンになる。ここで、このページ全体の表現は、本来手続き型言語では大

域変数に格納されるデータであることに注意が必要である。ここに、宣言型言語と手続き型言語の違いが端的に表れていると考えることができる。

各疑似関数について、それぞれ入力と出力の項のに注意して実関数を作成する。いくつかの実関数の簡単な仕様を示す。

edit-mode コマンドをキーボードから入力し、それに相当するオペレータを現在のページに付けて（append の場合 (`list 'append page '("abcdefg")`) など）返す。

print-page 現在のページを画面に出力する。

print-buffer 現在のミニバッファの内容を表示する。

load-page ファイル名を入力値として受けとり、そのファイルを読み込む。

save-page ファイル名と現在のページを受けとり、そのファイルにページの内容をセーブする。

repl-word 単語置換関数。目的の単語と置き換える単語と一行の文字列を入力値として受けとり、置換した文字列を返す。

以上の手順のように、項書換え系インタプリタを行エディタの等式仕様に関して部分計算し、その剰余プログラムに変更を加えることにより、手続き型言語 Scheme で記述した行エディタの実行可能コードが得られる。本研究ではさらにこのコードを、S-cheme から C への変換系にかけてコンパイルすることで、UNIX 上での実行可能ファイルを得ることを試みた。これにより、Scheme インタプリタでの実行より数倍程度速くなるので、規模の大きな仕様のプロトタイプに対して有効であると考えられる。

5 C 言語での実現との比較

例として取り上げた行エディタを直接 C 言語によってコーディングしてみた。本研究の手法を用いた場合との比較を表 1に示す。

C 言語で行エディタを書いた場合、文字列操作関数による文字列の処理が比較的複雑になる。その点、等式仕様で書いた場合は、データ構造の内部表現を意識する必要がないため、プログラミングの労力が軽減する。

また、本研究における手法により、正当性の検証、仕様が要求を満たしているかの確認などを行ないながらプログラムを作成していくことができ、ソフトウェア開発におけるラピッドプロトタイピングが可能となる。

	C 言語	本研究による手法
データ構造	データ構造を決定する必要がある (重つなぎリストにより記述)	既に等式中に規定されているので意識する必要はない
再利用性	データ構造、プログラム構成を十分に知る必要があるため再利用性はほとんどない	抽象度の高いレベルでの記述が可能であるので再利用が比較的容易。新たに等式を付加するだけで機能の拡張ができる
正当性	実際に実行する以外にプログラムの動作を確かめる方法がない	慣れればリストを見るだけで静的に正しさが実感できる。多ソート代数の定理証明系である *Larch-Prover*[12] などでの形式的検証も可能
デバッグ	かなりの労力が必要	項書換え系として記号的に実行できるので容易にできる
プログラムサイズ	プログラミングになれていなければ相当大きくなるおそれがある	わずか 30 行足らずの等式で同等の機能を実現でき、C 言語によるプログラムよりコンパクト

表 1: C 言語による行エディタプログラムと本研究による手法との比較

6 考察

本稿で提案した手法の特徴を以下にまとめる。

- 生成される手続き型プログラムのデータ構造は、インタプリタにおいて採用されたデータ構造がそのまま継承されるので、新たに考える必要はない。
- 等式仕様では等式が一つの単位となるので、もとの仕様に新たに等式が付加されたり一部が修

正されたりしても、全体を一から変換し直す必要はなく、漸増的 (incremental) な変換が可能である。

- 変換の効率、剰余プログラムの可読性を向上させることは、インタプリタを改良すること他ならないので、"良い" インタプリタを書くことに専念すれば良い。

今後の課題としては、

- Scheme の適当なサブセットについて、インタプリタを項書換え系で記述することにより、実関数も含めて等式論理上で形式的検証を行なうこと。
- 我々が文献 [9] に基づいて作成した代数的モジュール仕様合成器と組み合わせることにより、モジュールを単位としたプロトタイピングを試みること。
- *Similix* の最新版で可能になった、パターンマッチ、継続 (continuation) を用いてインタプリタを作成すること。これによりより効率のよい変換が可能になると考えられる。

などがある。

最後に、関連研究に触れておく。文献 [10] では、項書換え系を λ 計算に基づく関数型言語 Crescend へ変換する方法が取られているが、規則の適用の際に用いられるパターンマッチの制約から、重なりがなく線形な項書換え系のみを扱っている。本研究ではこうした制約は不要であり、また制約された項書換え系を扱う場合でも、インタプリタ中の単一化アルゴリズムを簡略化するだけで対処できる。文献 [11] においては、項書換え系を直接 C 言語のプログラムに変換する方法が取られているが、その変換アルゴリズムは本質的にインタプリタの部分計算を行なっているのと同じと考えられる。本研究でも、C 言語を対象とした部分計算器を用いれば、同様の変換コードが得られるはずである。行エディタの代数的仕様をこのシステムで実現した例が報告されている。

本研究では、項書換え系のインタプリタさえあれば、変換の正しさや対象とする項書換え系に対する制約を意識せずに変換が行なえ、また疑似関数と実関数の置き換えにより等式仕様の簡便な実現系となっている。

謝辞

最後に、我々の問い合わせに対し *Similix* の存在を教えて下さった Anders Bondorf 氏に感謝いたします。

参考文献

[1] 松本吉弘：ソフトウェア工学 (丸善, 1992), pp.55-70.

[2] I.Van Horebeek and J.Lewi : Algebraic Specification in Software Engineering (Springer-Verlag, 1989), pp.1-65.

[3] 二村良彦：部分計算, 古川康一・溝口文雄 共編：プログラム変換 (共立出版, 1987), pp.63-75.

[4] 二村良彦, 野木兼六：一般部分計算法, コンピュータソフトウェア, Vol.5, No.2 1988, pp.27-40.

[5] Anders Bondorf : Similix Manual system version 4.0, DIKU, Department of Computer Science, University of Copenhagen, 1991.

[6] 井田哲雄：計算モデルの基礎理論 (岩波書店, 1991), pp.224-297.

[7] 大須賀昭彦：項書換えシステムと完備化手続き, 井田哲雄編：新しいプログラミングパラダイム (共立出版, 1989), pp.166-186.

[8] H.Ehrig, B.Mahr : Fundamentals of Algebraic Specifications 1 (Springer-Verlag, 1985), pp.32-61.

[9] H.Ehrig, B.Mahr : Fundamentals of Algebraic Specifications 2 (Springer-Verlag, 1990).

[10] 布川博士, 黒田清隆, 富樫敦, 野口正一：項書き換え系の関数型言語への変換による実現, コンピュータソフトウェア, Vol.4, No.4 1987, pp.5-15.

[11] 酒井正彦, 坂部俊樹, 稲垣康善：抽象データ型の代数的仕様の直接実現系 Cdimple, コンピュータソフトウェア, Vol.4, No.4 1987, pp.16-27.

[12] Stephen J. Garland and John V. Guttag : A Guide to LP, The Larch Prover, Massachusetts Institute of Technology, 1991.