数理解析研究所講究録
1125 巻 2000 年 38-43

一般部分計算法(GPC)によるプログラム自動生成
Automatic Program Generation by Generalized Partial Computation

二村良彦[†] Yoshihiko FUTAMURA
小西善二郎[††] Zenjiro KONISHI
宋立トウ[†††] Litong SONG

[†]早稲田大学 理工学部 情報学科
School of Science and Engineering, Waseda University
[††]早稲田大学理工学総合研究センター
Advanced Rsearch Institute For Science And Engineering, Waseda University.
[†††]早稲田大学理工学研究科
Graduate School of Science and Enineering, Waseda University

概要

分かり易いが能率の悪いプログラムを入力すると，一流プログラマの手になるものに劣らない(桁違いに)高性能なプログラムを自動生産することが，一般部分計算法(GPC)の目的である．この桁違いな高速化を自動的に行うことにより，プログラムの生産性が飛躍的に向上することが期待できる．GPCはプログラムの自動高速化を行うために，推論機能 (定理証明器)，プログラム最適化に関する知識ベースおよび数式処理システム等を利用する．本稿では，GPCシステムで現在可能なプログラム自動生成の例，その限界および開 発スケジュール等について報告する．

1 はじめに

我々は，文献[3,4,5,9,10]で提案したアイデアに基づいてGPC実験システムを開発し[15,16,17]，高度なプログラム自動最適化が実現できることを示した．例えば，従来自動化が不可能であった下記のプログラム最適化が可能となった[13,14]:

例1：部分計算では従来プログラムの性能を桁違いに向上させることは不可能と考えられて来た[13]．しかし，我々のGPCでは，「ハノイの塔m手問題」のように，例えばディスクの枚数が1000枚の場合には約2^{1000}倍(厳密には，$O(2^n)$倍)高速化することを可能にすることを示せた．

例2：実用的なプログラムの自動高速化が可能なことも示せた：例えば，RSA等の暗号システムで使われる「冪乗の剰余」(mod(exp(n,m),d)，即ちnのm乗をdで割った余り)をnのm乗を計算せずに，直接計算するプログラムを自動生成することが可能となった．これにより，大きな数(n^m)の計算をせずに，小さな数(厳密にはd^2以下の数)の計算のみで，所望の結果を得ることができる(通常，nやmは100桁以上の整数であるのでn^mは100m桁以上の大きさになることに注意)．これは場合によっては1000倍以上の高速化になる．

例3：複雑な再帰プログラムを，自動的に簡略化できることを示せた：簡略化が容易でない再帰プログラムの古典的な例であるMcCarth yの91関数の自動簡略化を行うことができた．これは場合によっては10000倍以上の高速化になる．

このようなプログラムの桁違いの高速化は，一般プログラマにとっては殆ど不可能である．一般プログラマがこれを行えば，プログラム開発時間は数十倍以上に膨らみ，かつ正しいプログラムが得られることは期待できない．逆に，このようなプログラムの自動高速化が可能になれば，従来はプログラマとしての技量のない人たちが書いたプログラムでも，実用的性能のものに変換出来るようになる．これにより，より多くの人々がプログラム開発の作業に従事可能となる．

本稿では，GPCシステムで現在可能なプログラム自動生成の例，その限界および開発スケジュール等について報告する．

2 数式処理を利用したプログラム変換

現在我々のプログラム変換技術とREDUCE[2]等の数式処理システムを利用して実行可能なプログラム変換の一例を示す．文献[11]の297頁演習問題17bは，比較的難しい問題である．例えば，計算機科学のかなり優秀な学生にとってもその解

決は容易ではない．それは，次の再帰方程式を解けというものである．
(1.1) D(n,m)=0 if n<m or m≦0.
(1.2) D(n,n)=1.
(1.3) D(n,m)=(n-m)D(n-1,m)+D(n-1,m-1)
これをプログラム変換的アプローチでは次の様に解決する．先ず，
C(n,m)=D(n,m)/(n-m)! (n>m≧0) と置けば，
C(n,m)=D(n,m)/(n-m)! (定義)
=D(n-1,m)/(n-m-1)!+D(n-1,m-1)/(n-m)! **(unfold)**
=C(n-1,m)+C(n-1,m-1) **(fold)**
一方，C(n,m)=0(m>n or m≦0)かつC(n,n)=1.
従って，母関数[7,11]と2項定理を使えば，

$$G_n(z) = \sum_{m \geq 0} C_n(m) z^m$$

$$G_1(z) = C_1(1) = 1$$

$$G_n(z) = G_{n-1}(z) + z G_{n-1}(z) = (1+z) G_{n-1}(z)$$

$$= (1+z)^{n-1} G_1(z) = (1+z)^{n-1} = \sum_{m \geq 0} \binom{n-1}{m} z^m$$

従って，$C_n(m) = \binom{n-1}{m}$

従ってD(n,m)= $\binom{n-1}{m}$ (n-m)!
= (n-m)(n-1)(n-2)...(m+1)
この解法は，分かり易いのみならず，DやCのような形式の再帰方程式の解を求める際に応用できる汎用的なものである．母関数や2項定理を利用している以外は，GPCのプログラム変換の技術しか使用していない．このようなプログラム変換の自動化については[18]を参照されたい．

3 簡単なプログラム変換

我々は従来，本稿の導入部分で示したような，桁違いの高速化を実現するプログラム変換の例を示して来た．ここでは，より容易で分かり易い例を示す．先ず，リストxの要素を逆転させる関数rev(x)を次の様に定義する．

rev(x)=**if** x=[] **then** [] **else** ++(rev(cdr(x)),[car(x)]).

図1に示したようなGPCの結果，rev(++(u,[v]))は次のN1(u,v)に変換される．その計算中の++(append)は,再帰除去[14,19]により容易に除去可能であることに注意されたい．

N1(u,v)=**if** u=[] **then** [v] **else**
++(N1(cdr(u),v),[car(u)]).

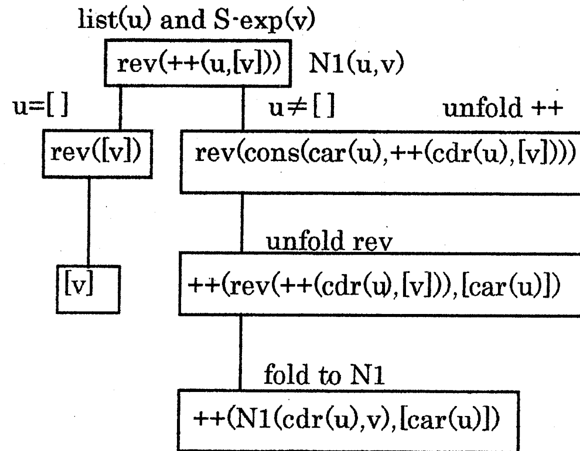

図1：rev(++(u,[v]))の部分計算過程(GPC木[3]による記述)

これ以降の節では，現在のGPC実験システムではまだ組み込んでいない，プログラム変換法について議論する

4 GPCにおける関数定義の一般化

プログラム変換は，与えられたプログラムとある意味で等価でかつより能率の良いプログラムを新たに定義する．その際に，新プログラムの定義域は元のプログラムのものと完全に一致させる必要がある．しかし，途中で生成される補助関数の定義域はプログラム変換システムで適当に決めることが出来る．従来のGPC[4,17]では，新しく定義される関数は全てGPC木の節であり，その定義域は節に対応するu情報(変数の値や性質に関する情報)により一意的に決められた．しかしそれでは，定義域が狭すぎて，畳み込みの対象からはずされる機会が増える．但し，逆に定義域を広げすぎると，特殊化の機会を失うことも起こりうる．従って，関数の定義域を現在のu情報よりも広げるか否かは非決定性の問題である．ちなみに，プログラム変換における一般化は[2]で最初に議論された．本稿では，新たに関数を定義する際に定義域を従来通りのものにするものと，それを出来るだけ拡張するもの(一般化)と両者を並列的に行い，結果の良い方を選択することを提案する．この追加的な関数定義はfoldingの際に行う．従ってfolding操作は，下記のように変更される．この一般化によりGPC木は複数個作成され，GPC森ができる．

畳み込み (Folding)操作
現時点で部分計算の対象となっているGPC木の節に対応する式eの部分式g(u)のうちで，GPC森の

節に含まれる式 (g')とunifiableで，かつgの定義域がg'の定義域に含まれているものを探す．このような式が複数個ある場合には，それ等の中で最長で，ある意味で，一番近い先祖を選びg'とする．またunificationによりuに対して代入された式をu'とする．g'の関数名をN(u)とすれば，gをN(u')で置きかえる．これは，複数の候補の中から1つ選ぶので，ヒューリスティックな選択である．どの選択が1番良いかは，やって見なければ分からないので，とりあえず仮名漢字変換のように最長一致のものを選んでやってみることにする．fold可能な式gが複数個含まれる場合には，全ての式をfoldしてみて，結果が1番良いものを選ぶ．fold出来るものが1つも見つからない場合には一般化を行う．

一般化(Genralization)

eの部分式gがfoldingに失敗したとする．そして，gの最左最内の非原始関数呼出しをh(k(u))とし，g=C[h(k(u))]とする．ただし，kは原始関数とする．この時，変数vの定義域を関数hの定義域としてC[h(v)]をGPCした結果をg'(v)とする．さらにg'(k(u))によりeの中のgを置換えた式をe'とする．この一連の操作をgの一般化と呼ぶ．

一般化と畳込みは抱き合わせで行われ，その後で展開 (unfolding) が行われることに注意されたい．一般化の例は第6節で示す．

5 タプリングとλ抽象

タプリング(tupling)は類似の計算を同時に行うことにより，プログラムの性能を大幅に向上させるプログラム変換技術であり，[2]で最初に明確な形で用いられた．その詳しい解説は[9]を参照されたい．GPCでは簡約化のフェーズにおいて次のようにしてタプリングとλ抽象を行う．

現時点で部分計算の対象となっているGPC木の節に対応する式eにその定義域の等しい2つの非原始関数呼出し$h_1(k_1(u))$と$h_2(k_2(u))$が含まれるとき，下記2ケースを試みる．但し，k_1とk_2は原始関数とする：

(5.1) タプリング： $k_1=k_2$かつ$h_1 \neq h_2$の時，$cons(h_1(v), h_2(v))$のGPCの結果をh(v)とする(これはfoldingの結果であっても良い)．ただし，変数vの定義域はh_1の定義域と同じとする．そして，e中の$h_1(k_1(u))$および$h_2(k_1(u))$をeに含まれない変数yに対して，各々car(y)およびcdr(y)で置換えて全体をe'とおく．その上でeを$(\lambda y.e')(h(k_1(u)))$とする．(注意：$k_1 \neq k_2$かつ$h_1=h_2$の時でも，$h_1k_1$と$h_2k_2$を上記の$h_1$と$h_2$と考える．さらに，$k_1$と$k_2$共に恒等関数と考えれば，ここで処理できる)．

(5.2) λ抽象： $k_1=k_2$かつ$h_1=h_2$の時，e中の全ての$h_1(k_1(u))$をeに含まれない変数yで置換えて全体をe'とおく．その上でeを$(\lambda y.e')h_1(u)$とする．

上記のような関数呼出しが3個以上含まれている場合も，上と同様にリストを用いてタプリングを行うことが出来る．但し，タプリングが有効になるのは，hを作る際に，それがfoldingによりうまく再帰関数になる場合に限られる．従って，同時に計算する非再帰関数呼出しの個数が増えるほど，成功の確率は減少する．

6 一般化，タプリングおよびλ抽象の例

定義通りの素朴なプログラムから高性能プログラムを作成する例を示す．問題は，「与えられた数列xに含まれる区間の中で最大の区間和を持つものを求めるプログラムmss(x)を作成せよ」である．ここでは計算量$O(n^3)$のプログラムを与えて，O(n)のプログラムを生成する．この問題自身は，プログラミングの教科書で古くから取り上げられている有名なものである．しかし，プログラム変換の問題として取り上げたのはBird[1]が最初であると考えられる．部分計算の例題としては[6]で用いられた．そしてこの問題が，全自動的に解決する目的で取り上げられたのは，本稿が最初であると考えられる．

以下ではGPCの過程はGPC木[3]により示される．ここではGPCの過程の図示をより簡単にするために，GPC木の枝として，2重線を追加した．これは，下線をつけられたW-redex[8,16]の展開形を示すためのものである．

6.1 素朴なプログラム

mss(x)=maxl(suml(segs(x)))．　但し，補助関数の定義は次の通りで，その計算量は$O(n^3)$である．

(1) segs(x)は数列xに含まれる全ての区間を列挙する関数である．

例：segs(NIL)=NIL, segs([4 3])=[[4] [4 3] [3]]
　　segs([1 4　3])=[[1] [1 4] [1 4 3] [4] [4 3] [3]]

segs(x)=**if** x=[] **then** [] **else** ++(inits(x), segs(cdr(x)))

(1.1) inits(x)は数列xの先頭の要素car(x)を含む全ての区間を列挙する．

例：inits(NIL)=NIL, inits([7])=[[7]], inits([4 3])=[[4] [4 3]]
　　inits([1 -2 3 1])=[[1] [1 -2] [1 -2 3] [1 -2 3 1]]

inits(x)=**if** x=[] **then** [] **else** cons([car(x)], distr(car(x), inits(cdr(x))))

(1.2) distr(a,b)はリストbの各要素の先頭にaをconsした要素のリストを作る．

例：distr(1, [[4] [4 3]])=[[1 4] [1 4 3]]

distr(a,b)=**if** b=[] **then** [] **else** cons([a.car(b)], distr(a,cdr(b)))

(2) suml(y)は，リストyの全ての要素毎の和のリス

トを作る.

例：suml(NIL)=NIL, suml([[4] [4 3] [3]])=[4 7 3]
　　suml([[1] [1 4] [1 4 3] [4] [4 3] [3]])=[1 5 8 4 7 3]

suml(y)=**if** y=[] **then** [] **else** cons(+n(car(y)),
　　suml(cdr(y)))

where +n([u1 u2 ... un])=u1+u2+...+un

(3) maxl(x)はリストx中の最大要素を取り出す.

例：maxl([1 -2 3 1])=3, maxl([])=-∞

maxl(x)=**if** x=[] **then** -∞ **else** max(car(x),
　　maxl(cdr(x)))

(4) 以上の諸関数に関して例えば次の様な性質が成立する．これは後ほど，GPCの簡略化フェーズ

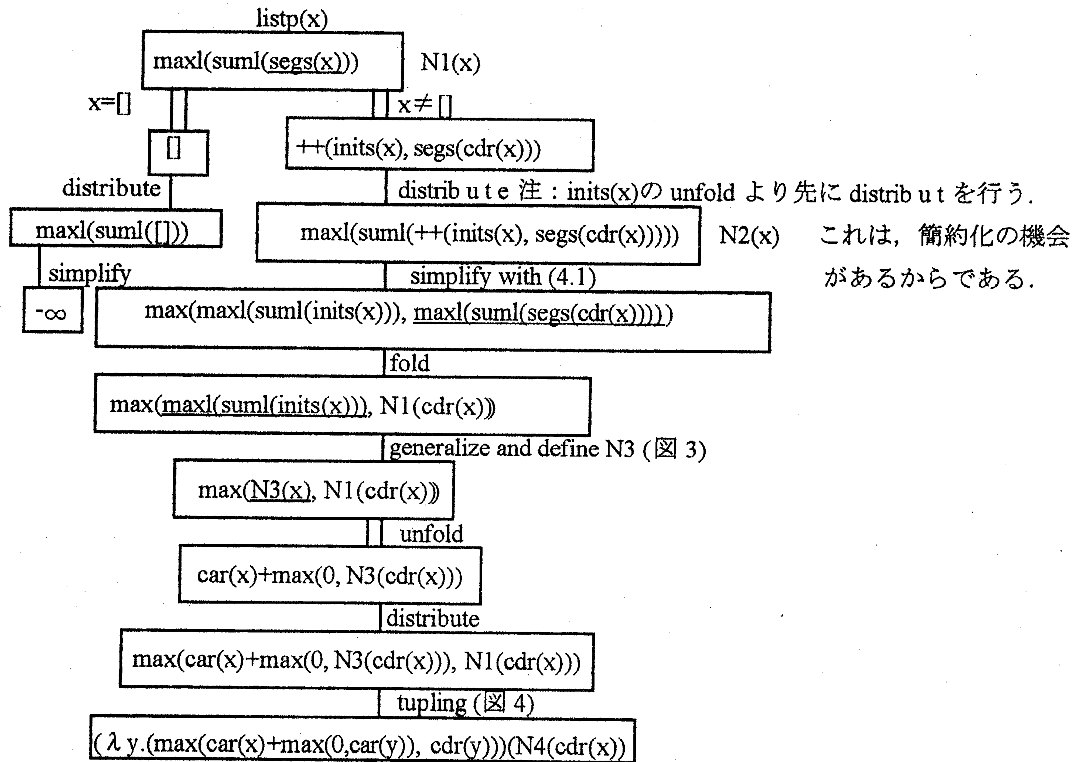

図2：xがリスト(listp(x))である場合のmaxl(suml(segs(x)))のGPC木

で利用される：

(4.1) maxl(suml(++(x,y)))=max(maxl(suml(x)), maxl(suml(y)))，(4.2) yがリストなら maxl(suml(cons([x],distr(x,y))))= max(x, x+maxl(suml(y)))，(4.3) max(a, a+b)=a+max(0,b)

6.2 GPCによる最適化

先ず，xがリスト(listp(x))である場合のmaxl(suml(segs(x)))のGPC木を作る(図2,3)．これより，N1およびN3の定義は下記のようになる：

N1(x)=**if** x=[] **then** -∞
　　else max(car(x)+max(0,N3(cdr(x))), N1(cdr(x))).

N3(x)=**if** x=[] **then** -∞ **else** car(x)+max(0,N3(cdr(x)))

ここで，N3は定義域がnilの場合も含むように一般化されていることに注意されたい．N1(x)の定義式中のN3とN4(図3)の呼出しがタプリングにより次の様に変換される．

N1(x)=**if** x=[] **then** -∞
　　else (λy.(max(car(x)+max(0,car(y)), cdr(y)))
　　　　(N4(cdr(x))

N4(x)=**if** x=[ ] **then** [-∞,-∞]
　　　　else (λy.(λz.([z.max(z, cdr(y))])
　　　　　(car(x)+max(0, car(y))) (N4(cdr(x))).

N4(x)の計算量は既にO(n)であるが，これは単純な線形再帰プログラムなので再帰除去が容易にでき，それによりメモリ使用量がO(1)の下記反復プログラムnew-mss(x)が得られる.

new-mss(x)=loop-mss(x,-∞,-∞).

loop-mss(x,u,v)=**while** x≠[ ] **do**
　begin u:=car(x)+max(0,u); v:=max(u,v); x:=cdr(x)
　end; return(v).

7 おわりに

定理証明器をフルに利用したGPCの基本部分は既に稼動している[15,17]．それにより期待通り高度なプログラム変換を行うことが出来る．本稿では，その基本部分だけで既に実行可能な例と，まだ対処できない一般化，タプリングおよびλ抽象

の3方式をGPCに組み入れる方法と，その例題を示した．これ等全てを自動的に実行できるGPCの開発が今後の課題である．

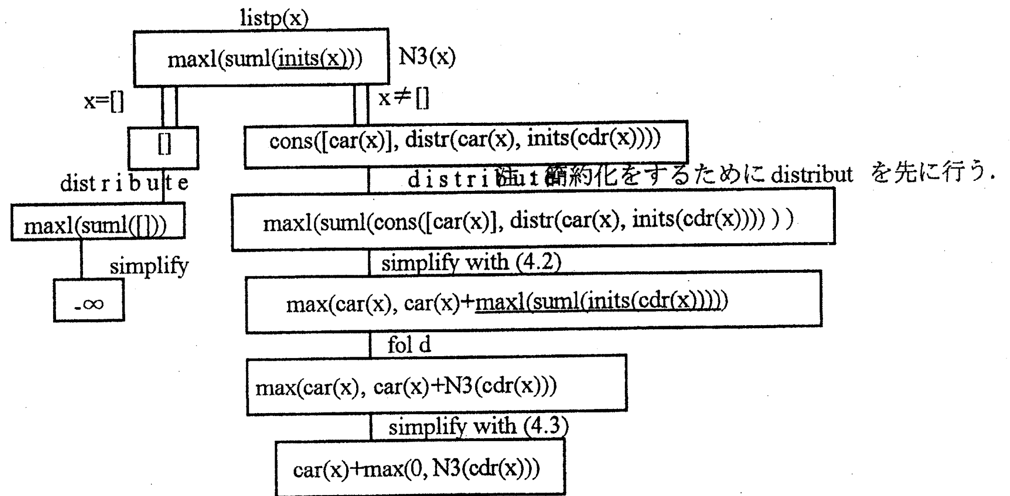

図3：maxl(suml(inits(x)))のlistp(x)に関するGPC木

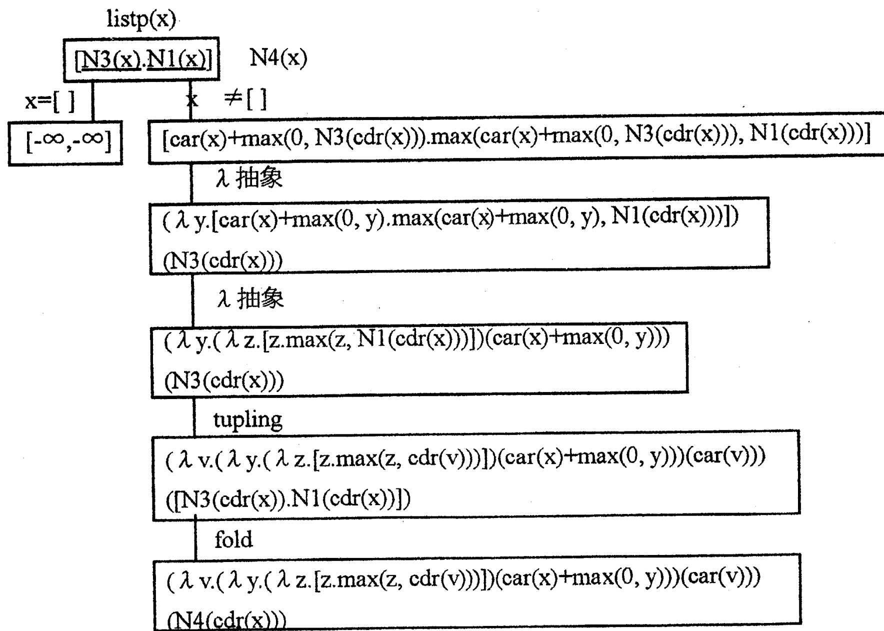

図4：タプリング(N3(x)とN1(x)の同時計算)に対するGPC木

参考文献

[1]Bird, R.S.: Algebraic identities for program calculation. Computer Journal, 32 (2), 1989, pp.122-126.

[2]Burstall, R.M. and Darlington, J. : A Transformation System for Developing Recursive Programs, JACM, Vol.24, No.1, 1977, pp.44-67.

[3]Futamura, Y., Nogi, K. and Takano, A.: Essence of generalized partial computation, Theoretical Computer Science 90 (1991) pp.61-79.

[4]Futamura, Y. and Nogi, K.: Program Transformation Based on Generalized Partial Computation, 5241678, US-Patent, Aug. 31, 1993.

[5]二村良彦，野木兼六： プログラム変換システム，特許2922207号，1999年7月19日．

[6]二村良彦： 一般部分計算の例， 日本ソフトウ

ェア科学会第10回大会，C5-3, 1993年11月.
[7]二村良彦，矢農正紀： 実際的プログラム変換の例， 日本ソフトウェア科学会第14回大会，1997年9月
[8]二村良彦,Song Litong,小西善二郎：一般部分計算(GPC)における制御構造と停止条件,日本ソフトウェア科学会大会論文集D5-1,1998年9月.
[9]Futamura, Y.: Partial Evaluation of Computation Process --- An approach to a Compiler-Compiler, Higher-Order and Symbolic Computation, Volume 12, December, 1999 .
[10]Futamura, Y.: Partial Evaluation of Computation Process Revisited, Higher-Order and Symbolic Computation, Volume 12, December , 1999.
[11]Graham, R.L., Knuth, D.E. and Patashnik, O.:*Concrete Mathematics*, Addison-Wesley, 1989.
[12] Hearn A.C.: REDUCE - A Case Study in Algebra System Development, *Lecture Notes on Comp. Science*, No. 144, Springer-Verlag, Berlin, 1982
[13]Jones, N. D.: An Introduction to Partial Evaluation, ACM Compting Surveys, Vol.28, No.3, September 1996, pp.480-503.
[14]Kakehi, K. and Futamura, Y.: Recursion removal under environments with cache and garbage collection, 京都大学数理解析研究所研究集会「プログラム変換と記号·数式処理」，1999年11月29日～12月1日.
[15]小西善二郎，二村良彦：一般部分計算（GPC）の実験システムの実装， 情報処理学会第58回全国大会論文集3K-05，1999年3月.
[16]小西善二郎，二村良彦：一般部分計算 (GPC) における停止条件の判定，日本ソフトウェア科学会大会論文集，1999年9月.
[17]小西善二郎，二村良彦：一般部分計算 (GPC) における定理証明系と停止条件の判定，京都大学数理解析研究所研究集会「プログラム変換と記号·数式処理」，1999年11月29日～12月1日.
[18]松谷将寛，二村良彦：数式処理を利用したプログラム変換，京都大学数理解析研究所研究集会「プログラム変換と記号·数式処理」，1999年11月29日～12月1日.
[19]Pettorossi, A. and Proietti, M.: Rules and Strategies for Transforming Functional and Logic Programs, ACM Computing Surveys, Vol.28, No.2, June 1996, pp.360-414.