数理解析研究所講究録
1125巻 2000年 59-64

一般部分計算 (GPC) における定理証明系と停止条件の判定

A theorem proving system and a terminating process for generalized partial computation (GPC)

小西 善二郎† 二村 良彦††
Zenjiro Konishi Yoshihiko Futamura

†早稲田大学 理工学総合研究センター
Advanced Research Institute for Science and Engineering, Waseda University
††早稲田大学 理工学部
School of Science and Engineering, Waseda University

概要

一般部分計算法 (GPC) は、分かりやすいが能率の悪いプログラムを、能率の良いプログラムに自動変換する手法である。この変換過程において、定理証明が重要な役割を果たす。停止条件の判定は、GPC システムの中で最も実現が困難な部分である。本稿では、始めに我々が採用した定理証明系を紹介し、それをどのように改造したのかを説明する。次に、GPC における停止条件を厳密に定義し、定理証明系を用いたその判定法を論ずる。最後に、現在いくつかのプログラムの自動変換に成功しているので、これについて報告する。

1 はじめに

我々は、プログラム変換に基づくプログラム自動生産システムの実現を目指している。その主要な基盤技術として、GPC (Generalized Partial Computation) [2] [4] を仮定している。

GPC におけるプログラム変換は、数種類の変換単位を基本とし、それらを何度も適用することによって最終的なプログラム (剰余プログラム) を得ている。それぞれの変換単位は、変換過程に応じたある特定の条件を満たすときに適用される。それらの条件を自動的に判定する手段として、我々は定理証明系を用いることにした。

多くの定理証明系は、時間を十分に費やして難しい定理を証明することを目的としている。一方、GPC に必要なのは簡単な定理を短時間で証明するものである。そこで我々は、良く知られた定理証明系を採用し、これを改造することによって GPC に特化した定理証明系を作成した。そして、これを呼び出す形で GPC 実験システムを実装した [5]。

実験システムでは、停止条件の自動判定がなかなか実現できなかった。本稿で言う停止とは、変換単位の一つである展開を行わないことである。GPC では、展開を行わない式はその他の変換単位も施されないので、その式の変換が停止するのである。この、展開を行う条件が非常に複雑であり、そのままでは定理証明系に入力できないことが問題であった。そこで我々は、この十分条件をうまく定め、さらに定理証明系とデータベースを併用することによって、ある程度の精度で停止条件が自動判定できるようにした [6]。

本稿では、現在のシステムで変換できるプログラムのうち、特に改善効果の大きい例を紹介する。(1) ハノイの塔問題の解の m 番目の操作を求める $O(2^n)$ 時間のプログラムを、$O(n)$ 時間のプログラ

ムに変換する [3]。(2) m^n を d で割った余りを求めるプログラムを、直接 m^n という大きな数を計算しないプログラムに変換する [4]。(3) マッカーシーの 91 関数を定数時間で計算するプログラムに変換する。

2 GPC における定理証明

本節では、GPC における変換単位のそれぞれについて説明し、そこで必要となる定理証明を明らかにする。なお、GPC の詳細については文献 [2] の通りなので省略する。

2.1 簡略化 (Simplification)

簡略化では、分岐除去に定理証明が必要である。これは、プログラム N が

$$N(x) = \textbf{if } P(x) \textbf{ then } E'(x) \textbf{ else } E''(x) \quad (1)$$

であるときに、条件 $x \in \mathrm{dom}(N) \to P(x)$ が成り立てば $N(x) = E'(x)$, 条件 $x \in \mathrm{dom}(N) \to \neg P(x)$ が成り立てば $N(x) = E''(x)$ と、プログラム N を変換することである。

2.2 分配 (Distribution)

プログラム (1) に対して、分岐除去ができない場合は分配を行う。これは、新たにプログラム N', N'' を用意し、$N'(x) = E'(x)$, $N''(x) = E''(x)$ とすることである。これらのプログラムの定義域は、

$$\begin{aligned} \mathrm{dom}(N') &= \mathrm{dom}(N) \cap \{x | P(x)\}, \\ \mathrm{dom}(N'') &= \mathrm{dom}(N) \cap \{x | \neg P(x)\} \end{aligned}$$

となるわけだが、これらの集合をなるべく単純に表す必要がある。定理証明系を用いて、この単純化を行う。なお、プログラム N については

$$N(x) = \textbf{if } P(x) \textbf{ then } N'(x) \textbf{ else } N''(x)$$

と変換し、これ以上は変換しない。

2.3 畳み込み (Folding)

プログラム N が $N(x) = g(h(k(x)))$ であり、もう変換しないプログラム N' が $N'(x) = h(x)$ であったとする。このとき、$N(x) = g(N'(k(x)))$ と変換する (畳み込む) ためには、条件

$$\{k(x) | x \in \mathrm{dom}(N)\} \subset \mathrm{dom}(N')$$

が成り立つことを確かめなくてはいけない。ここで定理証明を行う。

2.4 展開 (Unfolding)

もう畳み込みができないプログラムについては展開を試みる。プログラム N が $N(x) = g(h(k(x)))$ であるとする。プログラム h が、もしユーザ定義のものならば、条件

$$\{k(x) | x \in \mathrm{dom}(N)\} \subset \mathrm{dom}(h)$$

が成り立つことを定理証明系で確かめてから、h の定義にしたがって展開する。h がもう変換しないプログラムの場合は、これが W-redex [4] になることを確認してから展開する。展開できないプログラムは、それ以上変換しない。

2.5 再帰除去

簡略化、分配、畳み込み、そして展開による変換がすべて終了したら、最後に再帰除去を行う。プログラムを個々の再帰除去法に順次適用し、最終的なプログラムを得る。

3 GPC のための定理証明系

3.1 定理証明系 TPU

我々は、(1) 実績がある。(2) ソース・コードが公開されている。(3) Lisp で書かれている。などの理由により、GPC のための定理証明系として TPU [1] を採用した。ソース・コードが公開されているので改造が可能であるし、GPC システム全体が Lisp で書けるので自己適用などの実験に道がひらけるからである。

TPU は導出原理に基づく定理証明系である。これは、証明したい論理式の否定を公理系に追加して、そこから矛盾が導ければ証明されたとするものである。TPU における導出法は単位二元導出法と呼ばれるものであり、

- 台集合戦略 (set-of-support strategy)
- 関数の深さテスト (function depth test)
- 含意テスト (subsumption test)

という機能と併せて定理証明を行っている。

3.2 TPU の改造

一般に導出原理に基づく定理証明系は、公理が増えると計算時間が急激に増大する。この問題は、TPU においても避けられない。そこで我々は、公理系をなるべく小さくするため、TPU に等号調整 (paramodulation) [1] と単位書き換えという仕組

みを組み込んだ。

TPU で等式に関する定理証明を行うには、公理系に数多くの等号公理を追加しなくてはいけない。等号調整法を用いることにより、これらの等号公理を省くことができる。

単位書き換えとは、ひとつの項をそれに等しくてより短いものに置き換えたり、ひとつの原始論理式やリテラルをそれに同値でより短いものに置き換えたりする操作である。単位書き換えによって、例えば原始論理式 $a+1>6$ は $a>5$ に置き換わる。これを公理に基づいて生成しようとすると、論理式

$$x+y>z \leftrightarrow x>z-y \qquad (2)$$

と等式 $6-1=5$ を公理系に追加しなくてはいけない。式 (2) の左辺で y と z が数そのものである原始論理式を見つけたら、それを右辺のように置き換える。式 (2) の右辺で x と y が数そのものである原始論理式を見つけたら、それを左辺のように置き換える、といった書き換え規則を用意することにより、公理を減らすことが可能になる。

現在の実験システムにおける定理証明系は、これらの仕組みに基づいて、書き換え規則を 40 個用意し、公理を 11 個に押さえている。

その他、関数の深さテストや台集合戦略は行わないことにした。

以上の改造を TPU に施した結果、定理証明系のメイン・ルーチンは以下のようになった。

1. 証明したい論理式の否定を公理系に追加して、初期の節集合とする。
2. この節集合のうち単位節のみで証明を試みる。
   (a) 書き換え規則を用いて各単位節を同値でなるべく短いものに置き換える。
   (b) この単位節同士で矛盾が導ければ証明終了。そうでなければ次へ。
   (c) この単位節同士で等号調整を行い、新たに単位節を生成する。
   (d) 書き換え規則を用いて新しい単位節を同値でなるべく短いものに置き換える。
   (e) 新しい単位節に対して含意テストを行い、不要な単位節を削除する。
   (f) 古い単位節と新しい単位節とで矛盾が導ければ証明終了。そうでなければ次へ。
3. 初期の節集合から非単位節を一つ取り出す。(一度取り出された非単位節はもう使わない。) 非単位節がもうないなら証明失敗として終了する。
4. この非単位節とそれまでに得られた単位節とで節集合を作る。
5. この節集合で証明を試みる。
   (a) 非単位節と単位節の間で導出を繰り返し、新たに単位節を生成する。
   (b) 書き換え規則を用いて新しい単位節を同値でなるべく短いものに置き換える。
   (c) 古い単位節と新しい単位節とで等号調整を行い、さらに新たな単位節を生成する。
   (d) 書き換え規則を用いて新しい単位節を同値でなるべく短いものに置き換える。
   (e) 新しい単位節に対して含意テストを行い、不要な単位節を削除する。
   (f) 古い単位節と新しい単位節とで矛盾が導ければ証明終了。そうでなければ 3 へ。

3.3 定理証明系の制御 (1)

SDFU の各変換単位は、それぞれが施されるための条件を証明するために定理証明系を呼び出すわけであるが、条件をそのまま定理証明系に渡すのではなく、それを分解し、何度も定理証明系を呼び出すことによって全体を証明している。

例えば、畳み込みをするために、論理式

$$\{k(x)|\bigwedge_j R_j(x)\} \subset \{x|\bigwedge_j P_j(x)\} \qquad (3)$$

を証明しなくてはいけないとする。この場合は、各 j について節集合

$$\bigcup_i \{R_i(v)\} \cup \{\neg P_j(k(v))\}$$

を公理系に追加して定理証明系を呼び出し (v は新しい定数)、すべての j について矛盾が導ければ論理式 (3) が証明できたことになるのである。

4 停止条件の判定

4.1 W-redex の形式化

展開とは、部分計算の対象となる式において、そこに現れる関数呼び出しを、その関数の定義式に実引数を代入したものに置き換える操作である。展開は、しなければ部分計算に必要な情報が得られないが、必要以上にしても剰余プログラムをただ大きくするだけである。したがって、どのような関数呼び出しを展開するかが重要な問題になる。

GPC においては、W-redex である関数呼び出しを展開することにした。W-redex の厳密な定義

は以下に示されるが、関数呼び出しが W-redex であると、実引数の取り得る値の範囲がその関数の定義域より十分小さいので、展開すると特定的な式になり、部分計算に有用な情報が得られるだろうという意図がある。

プログラム $N'(u) = g(N(k(u)))$ の関数呼び出し $N(k(u))$ が W-redex であるとは、

$$\{k(y)|y \in \mathrm{dom}(N')\} \subset \mathrm{dom}(N)$$

であり、かつ

$$\begin{aligned}&(\exists i)(\forall x_0 \cdots x_{i-1}x_{i+1}\cdots x_{n-1})\\ &\quad [D_i = \emptyset \vee (0 < |D_i| < \infty \wedge E_i \neq \emptyset)\\ &\qquad \vee (|D_i| = \infty \wedge |E_i| = \infty)]\end{aligned} \tag{4}$$

を満たすことである。ここで、

$$\begin{aligned} D_i &= \{x_i|x \in \mathrm{dom}(N)\},\\ E_i &= \{x_i|x \in \mathrm{dom}(N)\\ &\qquad \setminus \{k_i(y)|y \in \mathrm{dom}(N')\}\},\\ k(y) &= (k_0(y), \ldots, k_{n-1}(y)) \end{aligned}$$

とする。

4.2 W-redex の十分条件

条件 (4) そのものを自動的に判定するのは非常に困難である。また、我々の定理証明系は、ある程度計算して証明が得られなければ処理を打ち切り、証明失敗と出力するので、本当は条件を満たすのにそれが分からない事が起り得る。W-redex ではないのに間違って展開することと、W-redex なのに間違って展開しないことのどちらが好ましいかを考えると、前者はいつまでも展開し続ける危険があるので、後者のほうが安全である。

そこで、まず (4) の十分条件で比較的判定しやすいものを定めた。

$$\begin{aligned}&(\exists i)[(\exists F : \text{有限集合})[\\ &\quad (\forall x_0 \cdots x_{i-1}x_{i+1}\cdots x_{n-1})[D_i \subset F]\\ &\quad \wedge (\exists a \in F)\\ &\qquad (\forall x_0 \cdots x_{i-1}x_{i+1}\cdots x_{n-1})[a \in E_i]]\\ &\quad \vee (\exists I : \text{無限集合})\\ &\qquad (\forall x_0 \cdots x_{i-1}x_{i+1}\cdots x_{n-1})[I \subset E_i]]\end{aligned} \tag{5}$$

そして、定理証明系にこの条件を入力し、証明に成功したら展開し、失敗したら展開しないことにした。条件 (4) と条件 (5) の違い、すなわち展開をあきらめる場合は次の通りである。

- ある $x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}$ に対して D_i が空集合になる場合。
- $x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}$ によって D_i が有限集合だったり無限集合だったりする場合。
- 任意の $x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}$ に対して D_i が有限集合であったとしても、
$$\bigcup_{x_0,\ldots,x_{i-1},x_{i+1},\ldots,x_{n-1}} D_i$$
が無限集合になる場合。
- 任意の $x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}$ に対して D_i が有限集合であり、かつ E_i が空集合でなかったとしても、
$$\bigcap_{x_0,\ldots,x_{i-1},x_{i+1},\ldots,x_{n-1}} E_i \tag{6}$$
が空集合になる場合。
- 任意の $x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}$ に対して E_i が無限集合であったとしても、集合 (6) が有限集合になる場合。

4.3 定理証明系の制御 (2)

SDFU の場合と同様に、W-redex の判定においても、条件を分解し、定理証明系を何度も呼び出すことによって全体を証明している。

例えば、

$$\{x|\bigwedge_j R_j(x)\} \cap \{k(x)|\bigwedge_j Q_j(x)\} = \emptyset \tag{7}$$

という論理式については、節集合

$$\bigcup_j \{R_j(k(v))\} \cup \bigcup_j \{Q_j(v)\}$$

を公理系に追加して定理証明系を呼び出し、矛盾が導ければ論理式 (7) が証明できたことになる。

関数 k を恒等関数とし、集合 $\mathrm{dom}(N), \mathrm{dom}(N')$, I をそれぞれ $\{x|\bigwedge_j P_j(x)\}$, $\{x|\bigwedge_j Q_j(x)\}$, $\{x|\bigwedge_j R_j(x)\}$ として論理式 (3), (7) を証明すれば、条件 (5) の部分 $I \subset E_i$ を確かめたことになる。他の部分についても同様である。

条件 (5) 全体を自動証明するには、存在量化子 $(\exists F$: 有限集合), $(\exists I$: 無限集合) の部分が問題になるが、これについては最後に触れる。

5 実現できたプログラム変換

5.1 ハノイの塔問題

ハノイの塔問題の解を s としたとき、s の m 番目の操作を求める計算を考える。まず、塔 a にある n 枚の円盤を、塔 b を利用してすべて塔 c に移動する全操作 s は次のプログラムで得られる。

$$\begin{aligned}&\mathrm{hanoi}(n, a, b, c) = \textbf{if } n = 1 \textbf{ then } [[a, c]]\\ &\qquad \textbf{else } [\mathrm{hanoi}(n-1, a, c, b),\end{aligned}$$

$[a,c]|\text{hanoi}(n-1,b,a,c)]$.

次に、$n-1$ 枚の円盤は $2^{n-1}-1$ 回の操作ですべて移動できることに注意すると、s から m 番目の操作を取り出すプログラムは次のように書ける。

$\text{move}(s,n,m) = \textbf{if } n=1 \textbf{ then } \text{car}[s]$
$\quad \textbf{else if } m=2^{n-1} \textbf{ then } \text{car}[\text{cdr}[s]]$
$\quad \textbf{else if } m>2^{n-1} \textbf{ then}$
$\quad\quad \text{move}(\text{cdr}[\text{cdr}[s]], n-1, m-2^{n-1})$
$\quad \textbf{else } \text{move}(\text{car}[s], n-1, m).$

したがって、

$\text{movehanoi}(n,m,a,b,c) =$
$\quad \text{move}(\text{hanoi}(n,a,b,c),n,m)$

が求めるプログラムになる。

プログラム movehanoi の計算量は、すべての操作の生成 hanoi に $O(2^n)$ 必要なので、全体としても $O(2^n)$ となる。GPC を用いれば、このような分かりやすいが能率の悪いプログラムを、能率の良いプログラムに自動変換できる。実際、$N_1(n,m,a,b,c) = \text{move}(\text{hanoi}(n,a,b,c),n,m)$ と定義して、実験システムに入力したところ、以下のような出力が得られた。

$N_1(n,m,a,b,c) = \textbf{if } n=1 \textbf{ then } [a,c]$
$\quad \textbf{else if } m=2^{n-1} \textbf{ then } [a,c]$
$\quad \textbf{else if } m>2^{n-1} \textbf{ then}$
$\quad\quad N_1(n-1,m-2^{n-1},b,a,c)$
$\quad \textbf{else } N_1(n-1,m,a,c,b).$ (8)

これは、$O(n)$ で計算できるプログラムである。

普通の部分計算は、入力パラメタに関する部分的な情報に基づいてプログラムを変換する。GPC はプログラム自身の構造に基づいて変換するので、入力パラメタがまったく与えられなくてもプログラムが変換できる。実は、いくつかの入力パラメタを特定すると、GPC はより強力な変換を行うようになる。

ハノイの塔問題で、特に 16 番目の操作を求める問題を考える。実験システムに定義 $N_1(n,a,b,c) = \text{move}(\text{hanoi}(n,a,b,c),n,16)$ を入力したところ、出力としてプログラム

$N_1(n,a,b,c) = \textbf{if } 5=n \textbf{ then } [a,c]$
$\quad \textbf{else } N_1(n-1,a,c,b)$ (9)

が得られた。これをよく見ると、次のような大変能率の良い $(O(1))$ プログラムが導ける。

$\textbf{if } \text{Odd}[n] \textbf{ then } [a,c] \textbf{ else } [a,b].$

場合によっては、GPC は自明な変換を行う。ハノイの塔問題で、特に円盤が 3 枚の場合を考える。定義 $N_1(m,a,b,c) = \text{move}(\text{hanoi}(3,a,b,c),3,m)$ のもとで、実験システムは以下のようなプログラムを構成した。

$N_1(m,a,b,c) = \textbf{if } m=4 \textbf{ then } [a,c]$
$\quad \textbf{else if } m>4$
$\quad\quad \textbf{then if } m=6 \textbf{ then } [b,c]$
$\quad\quad\quad \textbf{else if } m>6 \textbf{ then } [a,c]$
$\quad\quad\quad \textbf{else } [b,a]$
$\quad \textbf{else if } m=2 \textbf{ then } [a,b]$
$\quad \textbf{else if } m>2 \textbf{ then } [c,b]$
$\quad \textbf{else } [a,c].$ (10)

これは確かに能率が良い $(O(1))$ が、変換の内容は単に展開を繰り返しただけである。

実験システムは、自分の出力プログラムを再び入力プログラムとして扱うことができる。この性質を用いると、一部の入力パラメタが与えられた場合の変換に、もうひとつの方法がとれる。つまり、始めに与えられたパラメタを無視して変換し、そしてその結果に対してそのパラメタに基づいた変換をするのである。ハノイの塔問題の場合なら、16 番目の操作を求めるプログラムは、変換されたプログラム (8) を $m=16$ としてさらに変換しても得られるし、円盤が 3 枚の場合も同様にプログラム (8) を $n=3$ としてさらに変換すると得られる。実験システムでこれを行ったところ、それぞれプログラム (9), (10) が出力され、直接変換したのと同じ結果になった。

5.2 modexp 関数

modexp 関数は、m の n 乗を d で割った余りを求める関数である。ここで、べき乗は以下のように計算することにする。

$\exp(m,n) = \textbf{if } n=0 \textbf{ then } 1$
$\quad \textbf{else if } \text{Odd}[n] \textbf{ then}$
$\quad\quad m * \text{sqr}[\exp(m,(n-1)/2)]$
$\quad \textbf{else } \text{sqr}[\exp(m,n/2)].$

定義 $N_1(m,n,d) = \text{mod}[\exp(m,n);d]$ のもとで、実験システムは次のようなプログラムを構成した。

$N_1(m,n,d) = \textbf{if } n=0 \textbf{ then } 1$
$\quad \textbf{else if Odd}[n] \textbf{ then } \text{mod}[\text{mod}[m;d] *$
$\quad\quad \text{mod}[\text{sqr}[N_1(m,(n-1)/2,d)];d];d]$
$\quad \textbf{else } \text{mod}[\text{sqr}[N_1(m,n/2,d)];d].$

このプログラムは、m^n という大きな数を扱わなくなるので、より能率良く計算できる。

5.3 マッカーシーの 91 関数

マッカーシーの 91 関数は、

$$f(x) = \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } f(f(x+11))$$

と定義されるプログラムである。これは、

$$\mathrm{newf}(x) = \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } 91$$

と同じ関数を計算する。プログラム f は複雑な再帰呼び出しが起るので能率が悪いが、newf は定数時間で計算できる。

GPC に従えば、f は newf に自動変換できる。実際、実験システムでプログラム $N_1(x) = f(x)$ に対して GPC を行ったところ、次のようなプログラムが得られた。

$$\begin{aligned}
&N_1(x) = \textbf{if } x > 100 \textbf{ then } x - 10 \\
&\quad \textbf{else } N_3(x), \\
&N_3(x) = \textbf{if } x > 89 \\
&\quad \textbf{then if } x > 99 \textbf{ then } 91 \\
&\qquad \textbf{else if } x > 98 \textbf{ then } 91 \\
&\qquad \textbf{else if } x > 97 \textbf{ then } 91 \\
&\qquad \textbf{else if } x > 96 \textbf{ then } 91 \\
&\qquad \textbf{else if } x > 95 \textbf{ then } 91 \\
&\qquad \textbf{else if } x > 94 \textbf{ then } 91 \\
&\qquad \textbf{else if } x > 93 \textbf{ then } 91 \\
&\qquad \textbf{else if } x > 92 \textbf{ then } 91 \\
&\qquad \textbf{else if } x > 91 \textbf{ then } 91 \\
&\qquad \textbf{else if } x > 90 \textbf{ then } 91 \\
&\qquad \textbf{else } 91 \\
&\quad \textbf{else } f(N_3(x+11)).
\end{aligned}$$

現在の実験システムには再帰除去の機能がないので、これ以上は処理できない。再帰除去を行う場合は、

$$\begin{aligned}
N_3(x) &= \textbf{if } x > 89 \textbf{ then } 91 \\
&\qquad \textbf{else } f(N_3(x+11)) \\
&= f^{(1+\lfloor (89-x)/11 \rfloor)}(91) \\
&= 91 \quad (f(91) = 91)
\end{aligned}$$

となり、最終的にプログラム newf が得られる。

6 おわりに

前節のプログラムに対するプログラム変換の所要時間と定理証明系の呼び出し回数を表 1 に示す。定理証明系の呼び出し回数が多いのは、$x = 1 \to x = 1$ のような自明な定理も数えているからである。また、括弧内は停止条件の判定の内数である。展開すべき条件が証明できなければ停止すること、そしてその展開条件が複雑であることから、あまり展開しない場合ほど停止条件の判定に時間がかかることが分かる。

入力プログラム	変換の所要時間 [秒]		呼び出し回数	
ハノイの塔	61.4	(44.0)	83	(44)
(16 番目直接)	9.5	(6.4)	34	(17)
(16 番目二段階)	70.2	(50.2)	115	(61)
(3 枚直接)	43.5	(0.1)	103	(0)
(3 枚二段階)	89.6	(43.8)	162	(44)
modexp 関数	62.8	(51.1)	62	(30)
91 関数	94.0	(14.0)	305	(130)

表 1 プログラム変換の所要時間と定理証明系の呼び出し回数 (Pentium II 366MHz, Windows 98, Allegro Common Lisp 5.0.1, 括弧内は停止条件の判定の内数)

停止条件に関する今後の課題として、条件 (5) における有限集合と無限集合をどのように発見するかがある。現在の実験システムは、有用な集合をあらかじめデータベース化し、そこから一つづつ取り出して定理証明系に入力している。この方法では、よく知られた場合以外はまったく展開しなくなる。定義域と関数呼び出しの形から、動的に候補となる集合を生成する手法を見つける必要がある。

参考文献

[1] Chang, C. and R. C. Lee: *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

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

[3] 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 回大会論文集, 1993, pp. 317–320.

[4] 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第 15 回大会論文集, 1998, pp. 313–316.

[5] 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 回全国大会講演論文集, 1999, vol. 1, pp. 309–310.

[6] 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第 16 回大会論文集, 1999, pp. 309–312.