弱逆関数の自動導出によるプログラムの並列化

Automatic Weak Inversion for Parallelization

森田 和孝, 森畑 明昌, 胡 振江, 武市 正人
Kazutaka MORITA, Akimasa MORIHATA, Zhenjiang HU, Masato TAKEICHI
東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo
{Kazutaka_Morita,Akimasa_Morihata,hu,takeichi}@mist.i.u-tokyo.ac.jp

並列プログラムの構成は逐次プログラムに比べて難しいため, 逐次プログラムを自動的に並列化する手法が求められている. そこで本論文では弱逆関数という概念を導入し, 弱逆関数に基づいた自動的な並列化手法を提案する. 弱逆関数とは関数の出力から元の入力のうちのひとつを返す関数であり, 第三準同型定理 [12] と弱逆関数によって並列化を行うことができる. しかし弱逆関数を自動的に導出する手法は提案されていない. 本論文では弱逆関数を自動的に導出する手法を提案し, 第三準同型定理を用いた並列化が自動的に行えることを示す.

1 はじめに

大規模な問題を解く上で，並列計算は大きな役割を果たしてきた. 近年, 扱うデータの巨大化によりこのような大規模な問題を解く機会はますます多くなっている. また PC クラスタなどの並列計算機環境も身近なものになってきており, 並列計算はさらに重要性を増している. しかし並列実行可能なプログラムの構築は逐次プログラムの構築に比べて難しい. なぜなら, 効率的な並列計算のためには処理を独立に実行可能な部分問題に適切に分割しなければならないからある. このような理由のため, 並列プログラミングは非常に敷居の高いものとなっている. この問題を解決するための手法の一つとして, 逐次プログラムから並列プログラムを導出する手法に関する研究がこれまで行われてきた [5, 6, 10, 11, 12, 15, 16].

リストを走査する並列実行可能なプログラムの一般形のひとつとして, リスト準同型 [3] というものがある. リスト準同型のプログラムは, すべての要素に対する独立な計算とその結果をひとつにまとめる計算とで実現でき, 様々な並列計算機環境で効率良く実行することができる. 一方, ある特定の 2 種類の形で定義された逐次プログラムは第三準同型定理 [12] によってリスト準同型の形の定義を持つことが示されている. しかし第三準同型定理はリスト準同型の形の形の定義を持つということを示すのみで, そのリスト準同型の定義をどのように自動的に構成するかということは知られていなかった.

本論文では, 第三準同型定理を用いた関数型プログラムの自動的な並列化手法を提案する. そこでまず弱逆関数という新しい概念を導入する. 弱逆関数とは関数の出力から元の入力のうちのひとつを返す関数であり, 弱逆関数が分かれば第三準同型定理によって並列プログラムを自動的に導出できる. 弱逆関数を導出することは一般には困難であるが, 我々は対象とする言語に制限を加え, その言語上で弱逆関数の自動的な導出を示す. 実際に我々は弱逆関数を導出するシステムを実装した. また, この並列化手法にしたがって maximum prefix sum, maximum segment sum [2] といった古典的問題に対して, 効率的な並列プログラムが逐次プログラムから自動的に導出されることを示す.

本論文の構成は次の通りである. 続く第 2 節では, 本論文で用いられる表記法について簡単に述べる. 第 3 節では, 並列プログラムの一般形のひとつとしてリスト準同型について説明する. 第 4 節で弱逆関数という概念を紹介し, 第 5 節でその自動的な導出法について述べる. 弱逆関数を用いたプログラムの並列化については第 6 節で示す. 最後に, 第 7 節で関連研究について触れ, 本論文のまとめと今後の課題について第 8 節で述べる.

2 表記法

まず最初に本論文全体に共通の用語および表記法について説明する． 本論文の記法は関数型言語 Haskell [4] に基づいている.

2.1 関数型プログラム

関数型言語では, プログラムは関数定義の集合として構成される. プログラムの実行は, 関数を引数に適用しその結果を計算することにより行われる. 本論文では関数適用 $f(x)$ を $f\ x$ と空白を用いて表わし, 括弧は表記しない. 以下関数表記の慣習と演算子の定義について説明する. 一般的な二項演算子を表すものとして, 本論文では $\oplus$, $\otimes$, $\odot$ などを用いることにする. 関数適用は左側に優先的に結合し, $f\ a\ b$ は $(f\ a)\ b$ を意味する. また関数適用の結合順位は最も強く, $f\ x \oplus y$ は $f\ (x \oplus y)$ ではなく $(f\ x) \oplus y$ として解釈される. 関数合成の演算は $\circ$ によって表され, $(f \circ g)\ x = f\ (g\ x)$ である. 演算子 $\triangle$ は $(f\ \triangle\ g)\ x = (f\ x,\ g\ x)$ と定義される. 演算子 $\uparrow$ は 2 値の最大をとる演算を表現し, $x \uparrow y =$ **if** $(x \geq y)$ **then** x **else** y と定義される.

2.2 リスト

本論文ではリスト上で定義された関数のみを考える. リストは線形に順序のついた値の集まりであり, 例えば 1, 2, 3 の 3 つの要素からなるリストは $[1,\ 2,\ 3]$ と表す. 演算子 $+\!\!+$ はリストの連結を表し, $[1,\ 2,\ 3] +\!\!+ [4,\ 5,\ 6]$ は $[1,\ 2,\ 3,\ 4,\ 5,\ 6]$ である. この $+\!\!+$ は結合性をもつ演算である. また, 関数 $[\cdot]$ は任意の値値 a を取り, 要素 1 つのリスト $[a]$ を返す.

リストを処理する関数はパターンマッチを用いて定義される. 以下はリストの要素和を求める関数 sum の定義である.

例 2.1.

$$\begin{array}{lcl} sum\ [a] & = & a \\ sum\ ([a] +\!\!+ x) & = & a + sum\ x \end{array}$$ □

ここで a はリストの要素, x はリストを表している. 左辺に現れる $[a]$ や $([a] +\!\!+ x)$ がパターンになっており, $[a]$ は要素 1 つのリストに, $([a] +\!\!+ x)$ は要素 2 つ以上のリストにマッチする. マッチングが成功すると右辺が評価され, それが関数適用の結果として返される. 関数 sum の例では, 引数の要素が 1 つであるとき第 1 式のプログラムが, 引数の要素が 1 つより多い場合は第 2 式のプログラムが実行される. 第 2 式に引数がマッチした場合, 引数の先頭要素を a, 残りの部分を x として右辺が実行される.

例 2.2.

$$\begin{array}{lcl} sum\ [a] & = & a \\ sum\ (x +\!\!+ y) & = & sum\ x + sum\ y \end{array}$$ □

この例も例 2.1 と同様に引数の要素が 1 つであるとき第 1 式のプログラムが, 引数の要素が 1 つより多い場合は第 2 式のプログラムが実行される. しかし第 2 式の場合, $+\!\!+$ は結合的であるため引数が x と y に分割される場所は一意に定まらない. 本論文ではこの様な分割箇所が一意に定まらないパターンを持つ関数は分割箇所にかかわらず等しい値を返さなければならないものとする. 実際, 関数 sum は $+$ の結合性により分割箇所によらず等しい値となる.

3 プログラムの並列化

例 2.1 の形で定義された sum はリストの要素を先頭から順に計算していくため, このままでは効率的に並列処理できない. 一方, 例 2.2 の形で定義された sum は, その定義から部分問題へ分割できることが保証されているので, 並列処理に適している. そこで例 2.2 のような並列処理可能な関数定義の一般形として, リスト準同型 [3] というものを考えることにする.

3.1 リスト準同型

定義 3.1 (リスト準同型 [3])**.** リスト準同型は

$$\begin{array}{lcl} h\ [a] & = & f\ a \\ h\ (x +\!\!+ y) & = & h\ x \odot h\ y \end{array}$$

と定義される関数である. ただし $\odot$ は結合的な演算子とする. 以下, 関数 h が上で示されたリスト準同型によって定義される時 $h = (\!|f,\ \odot|\!)$ と表記する. □

関数 h がリスト準同型の形で与えられていると h はすべての要素に対する独立な計算とその結果をひとつにまとめる計算とで実現でき, 様々な並列計算機環境で効率良く実行することができる.

例 3.1. 関数 sum のリスト準同型の形の定義は

$$sum = (\!|id,\ +|\!)$$

である. ここで id は引数の値をそのまま返す恒等関数である. □

リスト準同型は並列プログラムの定義として便利な形であるが, 一般の関数はこのような形で書けるとは限らない. しかし多くのリスト上の関数は適切に組化することによりリスト準同型として定義できることが知られている [6]. そこで表現力を増すために組化リスト準同型というものを定義する.

定義 3.2 (組化リスト準同型 [6])**.** 組化リスト準同型を

$$\pi_1 \circ ([\![f,\ \odot]\!])$$

と定義する．ここで π_1 は射影関数であり, 組化された値のうち先頭の値を返す関数である． □

組化リスト準同型の計算のほとんどは組化されたリスト準同型で行われており, 組化リスト準同型は並列に計算することができる．よって組化リスト準同型はリスト準同型を含む一般的な並列計算可能な関数群を表現していると考えることができる．

例 3.2. Maximum prefix sum とはリストの先頭から始まる部分リストの要素和の最大値を求める問題である．よって, maximum prefix sum の解を求める関数 mps は, 例えばリスト $[1,\ 2,\ -1]$ に対して,

$$\begin{aligned} & mps\ [1,\ 2,\ -1] \\ & \quad = \quad 0 \uparrow 1 \uparrow (1+2) \uparrow (1+2+(-1)) \\ & \quad = \quad 3 \end{aligned}$$

となる．

この mps は sum と組化し

$$mps = \pi_1 \circ (mps \ \triangle \ sum)$$

と定義することにより組化リスト準同型となることが知られている [6]. 実際, $(mps \ \triangle \ sum)$ は

$$\begin{aligned} & (mps \ \triangle \ sum)\ [a] \\ & \quad = (a \uparrow 0,\ a) \\ & (mps \ \triangle \ sum)\ (x +\!\!+ y) \\ & \quad = (mps\ x \uparrow (sum\ x + mps\ y),\ sum\ x + sum\ y) \end{aligned}$$

と定義できるので,

$$\begin{aligned} & (mps \ \triangle \ sum) = ([\![f,\ \odot]\!]) \\ & \quad f\ a \qquad\qquad\qquad = (a \uparrow 0,\ a) \\ & \quad (x_p, x_s) \odot (y_p, y_s) \ = \ (x_p \uparrow (x_s + y_p),\ x_s + y_s) \end{aligned}$$

とリスト準同型の形で定義できる． □

例 3.3. Maximum segment sum とはリストの部分リストの要素和の最大値を求める問題である．よって, maximum segment sum の解を求める関数 mss は例えば,

$$\begin{aligned} & mss\ [-1,\ 2,\ 1] \\ & \quad = \quad 0 \uparrow -1 \uparrow 2 \uparrow 1 \\ & \qquad \uparrow (-1+2) \uparrow (2+1) \uparrow (-1+2+1) \\ & \quad = \quad 3 \end{aligned}$$

となる．この mss は mps, mts, sum と組化し

$$mss = \pi_1 \circ (mss \ \triangle \ mps \ \triangle \ mts \ \triangle \ sum)$$

と定義することにより組化リスト準同型となることが知られている [6]. ここで mts はリストの末尾で終わる部分リストの要素和の最大値を求める関数である．実際,

$$\begin{aligned} & (mss \ \triangle \ mps \ \triangle \ mts \ \triangle \ sum)\ [a] \\ & \quad = \quad (a \uparrow 0,\ a \uparrow 0,\ a \uparrow 0,\ a) \\ & (mss \ \triangle \ mps \ \triangle \ mts \ \triangle \ sum)\ (x +\!\!+ y) \\ & \quad = \quad (mss\ x \uparrow mss\ y \uparrow (mts\ x + mps\ y), \\ & \qquad mps\ x \uparrow (sum\ x + mps\ y), \\ & \qquad (mts\ x + sum\ y) \uparrow mts\ y, \\ & \qquad sum\ x + sum\ y) \end{aligned}$$

であるので, 例 3.2 と同様にしてリスト準同型の形で定義できる． □

しかし逐次プログラムからどのようにこのリスト準同型の定義を導出するかはあまり自明ではない．

3.2 第三準同型定理

プログラムをリスト準同型の形で記述するためには, 部分問題の結果から全体の問題の結果を計算する結合的な演算子 $\odot$ を見つける必要がある．しかしこれは一般に容易ではない．リスト準同型の導出を助ける定理に第三準同型定理 [12] がある．まず準備として, 左方向関数と右方向関数を定義する．

定義 3.3 (左方向関数)**.** 関数 h が任意の要素 a と任意のリスト x に対して,

$$\begin{aligned} h\ [a] \qquad & = \quad f\ a \\ h\ ([a] +\!\!+ x) \quad & = \quad a \oplus h\ x \end{aligned}$$

と定義できる時, h は左方向関数である． □

定義 3.4 (右方向関数)**.** 関数 h が任意の要素 a と任意のリスト x に対して,

$$\begin{aligned} h\ [a] \qquad & = \quad f\ a \\ h\ (x +\!\!+ [a]) \quad & = \quad h\ x \otimes a \end{aligned}$$

と定義できる時, h は右方向関数である． □

ここで $\oplus$ と $\otimes$ は結合的である必要はない．

定理 3.1 (第三準同型定理 [12])**.** 関数 h が左方向関数かつ右方向関数である時, またその時に限り, h は

リスト準同型の形で定義できる. つまり,

$$\begin{array}{lcl} h\ [a] & = & f\ a \\ h\ ([a] +\!\!+ x) & = & a \oplus h\ x \\ h\ (x +\!\!+ [a]) & = & h\ x \otimes a \end{array}$$
$$\Updownarrow$$
$$h \ = \ (\!|f,\ \odot|\!)$$

が成り立つ. □

補題 3.2 ([12])**.** 関数 g を $h \circ g \circ h = h$ を満たす関数とすると, 第三準同型定理の結合的な演算子 $\odot$ は

$$a \odot b = h\ (g\ a +\!\!+ g\ b)$$

である. □

ここで第三準同型定理の例を挙げる.

例 3.4. 関数 sum の左方向関数, 右方向関数としての定義は

$$\begin{array}{lcl} sum\ [a] & = & a \\ sum\ ([a] +\!\!+ x) & = & a + sum\ x \\ sum\ (x +\!\!+ [a]) & = & sum\ x + a \end{array}$$

であるので, sum はリスト準同型の形で定義できることが第三準同型定理よりわかる. 実際我々は例 3.1 でリスト準同型の形の定義を示した. □

例 3.5. 組化された関数 $(mps\ \triangle\ sum)$ は下に示す通り左方向関数かつ右方向関数である.

$$\begin{array}{l} (mps\ \triangle\ sum)\ [a] \\ \quad = \ (a \uparrow 0,\ a) \\ (mps\ \triangle\ sum)\ ([a] +\!\!+ x) \\ \quad = \ (0 \uparrow (a + mps\ x),\ a + sum\ x) \\ (mps\ \triangle\ sum)\ (x +\!\!+ [a]) \\ \quad = \ (mps\ x \uparrow (sum\ x + a),\ sum\ x + a) \end{array}$$

したがって, 第三準同型定理により $(mps\ \triangle\ sum)$ はリスト準同型の定義を持つ. 実際我々は例 3.2 でリスト準同型の形の定義を示した. □

しかし第三準同型定理はあくまでリスト準同型の形の定義の存在を示しているのみで, リスト準同型の形に必要な結合的演算子をどのように構成するかは述べていない. よって実際例 3.2 で示したようなリスト準同型としての結合的演算子をこの定義から導出することはあまり自明でない.

4 弱逆関数

本節では弱逆関数という概念を導入し, 弱逆関数が補題 3.2 の条件を満たすことを示す. そして弱逆関数を導出することにより, 補題 3.2 から結合的な演算子を得る.

関数 f の弱逆関数 g は f の出力から f の元の入力のうちの 1 つを返す関数であり, 次のように定義される.

定義 4.1 (弱逆関数)**.** 関数 g が

$$\forall y \in \mathrm{range}(f),\ g\ y = x \Rightarrow f\ x = y$$

という性質を満たす時, 関数 g を関数 f の弱逆関数と定義する. ただし $\mathrm{range}(f)$ は関数 f の値域を意味する. □

補題 4.1. 弱逆関数は任意の関数に少なくとも 1 つ存在する.

証明. $g\ t$ が $f\ x = t$ を満たす x を任意に返す関数とする. このとき弱逆関数の定義より明らかに g は f の弱逆関数のうちの 1 つである. よって弱逆関数は任意の f に対して少なくとも 1 つ存在する. □

関数 f が全単射の時に f の弱逆関数は逆関数になり, 逆関数は弱逆関数の特別な場合である.

また弱逆関数は一般に複数存在する. 例として sum の弱逆関数について考える. 要素の和が引数の値と同じになるリストを返す関数は全て sum の弱逆関数である. よって

$$\begin{array}{lcl} g_1\ a & = & [a] \\ g_2\ a & = & [a-1,\ 1] \\ g_3\ a & = & [1,\ 2,\ a-3] \\ g_4\ a & = & [a/2,\ a/2] \end{array}$$

と定義される関数 g_1, g_2, g_3, g_4 はいずれも sum の弱逆関数である.

以下, 関数 f° を関数 f の弱逆関数のうちのひとつとする.

弱逆関数の例として,

$$\begin{array}{lcl} sum^\circ & = & [\cdot] \\ mps^\circ & = & [\cdot] \\ sort^\circ & = & id \\ head^\circ & = & [\cdot] \end{array}$$

のようなものが挙げられる. ここで $sort$ は引数のリストの要素をソートしたリストを返す関数, $head$ は引数のリストの先頭要素を返す関数である.

弱逆関数は以下の性質により並列化に用いることができる.

補題 4.2. 弱逆関数は

$$f \circ f^{\circ} \circ f = f$$

という性質を満たす.

証明. 弱逆関数の定義より, $f \circ f^{\circ}$ は f の値域の値を適用しても同じ値を返す. よって $f \circ f^{\circ} \circ f = f$ である. □

定理 4.3. 関数 h が左方向関数かつ右方向関数である時,

$$\begin{aligned} h &= (\![f, \odot]\!) \\ a \odot b &= h\ (h^{\circ}\ a +\!\!+ h^{\circ}\ b) \end{aligned}$$

が成立する.

証明. 補題 3.2 と補題 4.2 より明らかである. □

よって関数 h が左方向関数かつ右方向関数である時, h の弱逆関数が求まれば h のリスト準同型による定義が得られる

5 弱逆関数の導出

リスト準同型の結合的な演算子を弱逆関数を用いて構成できることがわかったので, 弱逆関数を導出することを考える. 多くの関数は組化リスト準同型として定義できるが, 組化された関数の弱逆関数は個々の関数の弱逆関数がわかっても単純には求まらない. しかし多くの並列プログラムは組化リスト準同型によって記述されるため, 第三準同型定理を並列化に用いるためには組化された関数の弱逆関数を求める必要がある. 本節ではこのような関数に対して弱逆関数を求めるアルゴリズムを示す.

5.1 言語

本節では, 図 1 に示す言語により記述されたプログラムに対して弱逆関数を求めるアルゴリズムを提案する. プログラムはリストを走査し値を返す関数の定義 *def* の列によって構成されている. また *lterm* は線形な項であり, 条件分岐, 関数呼び出し, 加算と減算, 定数による乗算と除算により構成される. ここで *lterm* に含まれる関数 *fun* は必ずプログラム中で定義されていなくてはならない. プログラムは組化リスト準同型に含まれる関数を順に記述することに

prog	::=	*def* $\cdots$ *def*	(プログラム)
def	::=	*fun* $[a] = lterm$;	
		fun $([a] +\!\!+ x) = lterm$	(定義)
lterm	::=	*lterm addop lterm*	(加減算)
	|	*lterm mulop num*	(乗除算)
	|	*fun x*	(関数適用)
	|	*num*	(定数)
	|	*a*	(リストの要素)
	|	**if** (*cond*) **then** *lterm*	
		else *lterm*	(条件文)
cond	::=	*lterm relop lterm*	(比較演算)
	|	$cond \vee cond$	(論理和)
	|	$cond \wedge cond$	(論理積)
num	::=	$0 \mid 1 \mid 2 \mid \cdots$	
addop	::=	$+ \mid -$	
mulop	::=	$* \mid /$	
relop	::=	$= \mid < \mid \leq \mid > \mid \geq \mid \neq$	

図 1: 弱逆関数導出の対象とするプログラムの言語

より構成される. 例えば, 関数 $(mps \triangle sum)$ はこの言語では

$$\begin{aligned} mps\ [a] &= \textbf{if}\ (a \leq 0)\ \textbf{then}\ 0\ \textbf{else}\ a \\ mps\ ([a] +\!\!+ x) &= \textbf{if}\ (0 \leq a + mps\ x) \\ &\quad\ \textbf{then}\ a + mps\ x \\ &\quad\ \textbf{else}\ 0 \end{aligned}$$

$$\begin{aligned} sum\ [a] &= a \\ sum\ ([a] +\!\!+ x) &= a + sum\ x \end{aligned}$$

と記述できる. この言語で記述された関数の弱逆関数を導出することを考える.

5.2 弱逆関数の導出

本論文で提案する弱逆関数の導出法は次の手続きで構成される. まず, 弱逆関数が返すリストの長さを仮定して関数を展開し, 条件文と連立一次方程式の集合を得る. 次に得られた連立方程式を解き, 条件文とそれに対応する計算を得る. これが得られる弱逆関数である. さらに得られた弱逆関数の条件分岐を減らすことにより効率化を行う. 最後に得られた弱逆関数の定義域が正しいかどうかを検証する. これらの手続きを順に解説していく.

5.2.1 関数の展開

弱逆関数はその入力引数と出力リストの関係により定義される. 逆に言えば、入力引数と出力リストの関係が得られれば弱逆関数を得ることができる. 我々の提案するアルゴリズムでは, この関係を得るために, 弱逆関数の出力したリストを元の関数に適用すると入力引数を返すということを利用する.

例として maximum prefix sum を考える. いま, 関数 $(mps \vartriangle sum)^\circ$ が $(p,\ s) \in \mathrm{range}(mps \vartriangle sum)$ なる値 $(p,\ s)$ を入力として長さ 1 のリストを返すと仮定する. つまり,

$$(mps \vartriangle sum)^\circ\ (p,\ s) = [a]$$

とする. このとき, 弱逆関数の定義より,

$$(p,\ s) = (mps \vartriangle sum)\ [a]$$

である. よって弱逆関数の入力変数 $(p,\ s)$ と出力リスト $[a]$ の関係が以下のように得られる.

$$p = mps\ [a],\quad s = sum\ [a]$$

さらに関数 mps と sum の定義を展開すると以下の式が得られる.

$$p = \mathbf{if}\ (a \le 0)\ \mathbf{then}\ 0\ \mathbf{else}\ a,\quad s = a$$

ここで条件分岐を列挙して,

$$\begin{aligned} \{a \le 0\} &\Rightarrow p = 0,\ \ s = a \\ \{a > 0\} &\Rightarrow p = a,\ \ s = a \end{aligned}$$

という式を得る. この関係式は以下のように理解することができる. 弱逆関数 $(mps \vartriangle sum)^\circ$ は 2 入力変数 p, s の値が, (1) p が 0 かつ s が 0 以下, (2) p, s の値が等しくともに正値, である場合には長さ 1 のリスト $[s]$ を返せばよい.

以上の導出によって得られた結果は正しい. しかし，以上の導出では弱逆関数 $(mps \vartriangle sum)^\circ$ の部分的な定義, すなわち 2 入力変数 p, s の値が上記条件を満たす場合の定義, のみしか得られない. 理論上は加算無限を含む全てのリスト長について同様に計算すれば, 弱逆関数 $(mps \vartriangle sum)^\circ$ の定義が得られると考えられる. しかしこれは現実的ではない.

我々の提案するアルゴリズムでは, 有限の処理で弱逆関数を得るため以下の 2 点を仮定する. まず, 弱逆関数は常に高々定数長のリストを返すとする. 次に, 出力されるリストの要素の値は全て一意に定まるとする. 以上の仮定のもとでは, 定義されている関数の数を n として, 長さ 1 から n までのリストについて上記のように関数定義を展開し関係式を得れば十分である. なぜなら, 1 つのリストに対して関数定義を展開して得られる関係式は定義されている関数の数に等しく, これらの関係式から出力リストが一意に定まると仮定すると, 出力リストの要素数, すなわち求めるべき変数の数, は高々定義されている関数の数, すなわち関係式の数, でなければならないからである.

まとめるとアルゴリズムは以下となる. 1 から定義されている関数の数の長さまでのリストについて, 元の関数を用いて計算を行い, 入力変数と出力リストの要素に関する関係式を得る. なお, こうして得られた関係式は連立方程式と見ることができるが, そのまま解くには条件分岐が扱いづらい. そこで事前に全ての条件分岐を列挙することにより, 条件と連立一次方程式の集合を得る.

本節では 2 つの仮定により連立方程式の数を減らした. よってこの弱逆関数の定義域は狭くなっている可能性がある. 得られた弱逆関数の定義域については 5.2.4 節で議論する.

例 5.1. Maximum prefix sum を例にとって考える. 変数 p と s を

$$(p,\ s) \in \mathrm{range}(mps \vartriangle sum)$$

とする. 今回定義されている関数は mps と sum の 2 つであるので, 出力リストの長さは高々2 であるとする. 出力リストの長さが 1 の時に得られる関係式は

$$\begin{aligned} \{a \le 0\} &\Rightarrow p = 0,\ \ s = a \\ \{a > 0\} &\Rightarrow p = a,\ \ s = a \end{aligned}$$

であった. 同様に出力リストの長さが 2 である場合を考える. このとき, $(mps \vartriangle sum)^\circ\ (p,\ s) = [a,\ b]$ とすると,

$$(p,\ s) = (mps \vartriangle sum)\ [a,\ b]$$

である. よって $(p,\ s)$ と $(a,\ b)$ の間に

$$p = mps\ [a,\ b],\quad s = sum\ [a,\ b]$$

という関係が成り立つ. この連立方程式を解いて a, b を求めればよい. ここで関数を定義に従って展開すると,

$$\begin{aligned} p &= \mathbf{if}\ (b \le 0) \\ &\quad\ \mathbf{then\ if}\ (0 \le a)\ \mathbf{then}\ a\ \mathbf{else}\ 0 \\ &\quad\ \mathbf{else\ if}\ (0 \le a + b)\ \mathbf{then}\ a + b\ \mathbf{else}\ 0 \\ s &= a + b \end{aligned}$$

となる．しかしこの連立方程式を解くには条件分岐が扱いづらい．そこで事前に全ての条件分岐を列挙する．すると

$$\begin{array}{lll}
\{b \le 0 \wedge 0 \le a\} & \Rightarrow & p = a,\ \ s = a + b \\
\{b \le 0 \wedge 0 > a\} & \Rightarrow & p = 0,\ \ s = a + b \\
\{b > 0 \wedge 0 \le a + b\} & \Rightarrow & p = a + b,\ \ s = a + b \\
\{b > 0 \wedge 0 > a + b\} & \Rightarrow & p = 0,\ \ s = a + b
\end{array}$$

となり, 条件と連立一次方程式のペアの集合が得られる．

結局これらをあわせて,

$$\begin{array}{lll}
\{a \le 0\} & \Rightarrow & p = 0,\ \ s = a \\
\{a > 0\} & \Rightarrow & p = a,\ \ s = a \\
\{b \le 0 \wedge 0 \le a\} & \Rightarrow & p = a,\ \ s = a + b \\
\{b \le 0 \wedge 0 > a\} & \Rightarrow & p = 0,\ \ s = a + b \\
\{b > 0 \wedge 0 \le a + b\} & \Rightarrow & p = a + b,\ \ s = a + b \\
\{b > 0 \wedge 0 > a + b\} & \Rightarrow & p = 0,\ \ s = a + b
\end{array}$$

となる． □

5.2.2 連立方程式の求解

前節の処理により入力変数と出力リストの関係が連立一次方程式として得られた．よってこの方程式を解くことにより弱逆関数が得られる．ここで出力リストの要素の値が一意に定まらない場合は仮定に反するので無視する．連立方程式を解いて得られた値を条件の含まれている入力変数に代入することにより条件を求める．

例 5.2. Maximum prefix sum の場合, 得られた集合

$$\begin{array}{lll}
\{a \le 0\} & \Rightarrow & p = 0,\ \ s = a \\
\{a > 0\} & \Rightarrow & p = a,\ \ s = a \\
\{b \le 0 \wedge 0 \le a\} & \Rightarrow & p = a,\ \ s = a + b \\
\{b \le 0 \wedge 0 > a\} & \Rightarrow & p = 0,\ \ s = a + b \\
\{b > 0 \wedge 0 \le a + b\} & \Rightarrow & p = a + b,\ \ s = a + b \\
\{b > 0 \wedge 0 > a + b\} & \Rightarrow & p = 0,\ \ s = a + b
\end{array}$$

のうち第 4, 5, 6 式は a, b が一意に定まらないので無視する．よって

$$\begin{array}{lll}
\{a \le 0\} & \Rightarrow & p = 0,\ \ s = a \\
\{a > 0\} & \Rightarrow & p = a,\ \ s = a \\
\{b \le 0 \wedge 0 \le a\} & \Rightarrow & p = a,\ \ s = a + b
\end{array}$$

のそれぞれの方程式を解くと,

$$\begin{array}{lll}
\{a \le 0 \wedge p = 0\} & \Rightarrow & a = s \\
\{a > 0 \wedge p = s\} & \Rightarrow & a = p \\
\{b \le 0 \wedge 0 \le a\} & \Rightarrow & a = p,\ \ b = s - p
\end{array}$$

となる．ここで第 1, 2 式の解が求まるためには p と s に依存関係が必要なので, これを新しく条件に加えた．得られた結果から p と s による条件を求めて,

$$\begin{array}{lll}
\{s \le 0 \wedge p = 0\} & \Rightarrow & a = s \\
\{p > 0 \wedge p = s\} & \Rightarrow & a = p \\
\{p \ge s \wedge p \ge 0\} & \Rightarrow & a = p,\ \ b = s - p
\end{array}$$

という, 入力変数に対する条件とそれに対応する計算式が得られる．

よって弱逆関数を

$$\begin{array}{l}
(mps \vartriangle sum)^{\circ}(p,\ s) \\
\quad = \quad \textbf{if}\ (s \le 0 \wedge p = 0)\ \textbf{then}\ [s] \\
\qquad\quad \textbf{else if}\ (p > 0 \wedge p = s)\ \textbf{then}\ [p] \\
\qquad\quad \textbf{else if}\ (p \ge s \wedge p \ge 0)\ \textbf{then}\ [p,\ s - p]
\end{array}$$

と導出することができる． □

5.2.3 効率化

前節までのアルゴリズムにより得られた弱逆関数の効率は一般に良くない．なぜなら条件により表現された分岐の数は, 定義に含まれる関数の数を n, 各関数に含まれる条件分岐の数の和を m として, 最悪で $2^{m(n+1)}$ 個となるからである．そこで不要な分岐を削除することで分岐の数を減らす. 具体的には, 得られた弱逆関数の i 番目の条件分岐の条件を C_i としたときに

$$C_i \Rightarrow \bigvee_{i \neq k} C_k \tag{1}$$

が真なら, 条件 C_i を除いてもこの弱逆関数の定義域は変わらない．また, 入力が定義域内であれば全ての分岐は弱逆関数として正しい値を返す．よって C_i に対応する分岐は冗長であり, 取り除くことができる．式 (1) は Presburger 算術 [19] の形をしており, quantifier elimination という手続きで真偽の判定が行えることが知られている [7].

例 5.3. Maximum prefix sum の例では,

$$\begin{array}{lll}
\{s \le 0 \wedge p = 0\} & \Rightarrow & a = s \\
\{p > 0 \wedge p = s\} & \Rightarrow & a = p \\
\{p \ge s \wedge p \ge 0\} & \Rightarrow & a = p,\ \ b = s - p
\end{array}$$

において, $\{s \le 0 \wedge p = 0\}$ と $\{s \le 0 \wedge p = 0\}$ は $\{p \ge s \wedge p \ge 0\}$ に含まれるため, 入力値 p, s が第 1, 2 式の条件を満たす時は第 3 式を実行しても弱逆

関数として正しい値を返す．よって冗長な式を除くことにより条件分岐の数を減らすことが可能である．このとき

$$(mps \vartriangle sum)^{\circ}(p,\ s) = \textbf{if}\ (p \geq s \wedge p \geq 0)\ \textbf{then}\ [p,\ s-p]$$

と条件を 1 つにまとめることができる． □

5.2.4 定義域の正しさの検証

5.2.1 節で議論したように, 以上の処理により得られた弱逆関数は一般には部分関数となる．つまり, 得られた弱逆関数の i 番目の条件分岐の条件を C_i としたときに $\bigvee_i C_i = true$ は一般には成り立たない．しかし, 弱逆関数はその定義より元の関数の値域についてのみ適切な値を返せばよく, 全域関数である必要はない．よって元の関数の値域を条件分岐が覆っていればよい．つまり, 元の関数の値域を表現した条件を P としたとき

$$P \Rightarrow \bigvee_i C_i \tag{2}$$

が $true$ なら, この弱逆関数は確かに弱逆関数の定義を満たす．

例 5.4. 例 5.3 で得られた maximum prefix sum のプログラムについて考える．関数 mps の出力値は関数 sum の出力値と同じかそれ以上であるので, 任意のリスト x について $(mps \vartriangle sum)\ x = (p,\ s)$ としたとき $p \geq s \wedge p \geq 0$ が一般に成立する．よって式 (2) は成立するのでこの弱逆関数は必ず値を返す．以上より, 正しいプログラムであることが示された． □

5.3 弱逆関数自動導出システムの実装

我々は本節の手法を用いた弱逆関数の導出法を実装した．実装言語には C++言語を用いた．また, Presburger 算術の真偽判定には omega test [20] という手法を用いた Omega Library [1] を用いた．計算時間は, Presburger 算術の真偽判定の計算量が項の数を n として $O(2^{2^n})$ であるため, 条件分岐の効率化の箇所が主な部分を占める．しかし Omega Library は Presburger 算術の真偽判定を効率的に実装しており, 多くの場合多項式時間で計算可能である．また, 5.2.2 節で言及したように条件分岐の数も仮定を用いて事前に減らしている．結果, 我々のシステムは現実的な時間で弱逆関数を求めることができる．

例 5.5. 関数 sum の弱逆関数 f は

```
f (s) = [s]
```

と得られる． □

例 5.6. 関数 $(mps \vartriangle sum)$ の弱逆関数 f は

```
f (p, s) =
    if (s<=p && 0<=p) then [p, -p+s]
```

と得られる．この時の条件

$$sum\ x \leq mps\ x$$
$$0 \leq mps\ x$$

は任意の x に対して常に成り立つので, この弱逆関数は正しい． □

例 5.7. 関数 $(mss \vartriangle mps \vartriangle mts \vartriangle sum)$ の弱逆関数 f は

```
f (m, p, t, s) =
    if (0<=p<=m && 0<=t<=m && s+m<=t+p)
            then [p, -p-t+s, m, -m+t]
```

と得られる．この時の条件

$$0 \leq mps\ x \leq mss\ x$$
$$0 \leq mts\ x \leq mss\ x$$
$$sum\ x + mss\ x \leq mps\ x + mts\ x$$

は任意の x に対して常に成り立つので, この弱逆関数は正しい．この弱逆関数の導出に要した実際の実行時間は Pentium M 1.70GHz の CPU, 512MB のメモリを持った PC で time コマンドで計測して 20 秒程度である． □

6 弱逆関数を用いたプログラムの並列化

本論文で提案する並列化手法は次の手続きによって行われる．まず, 左方向関数, 右方向関数としての 2 つの逐次プログラムの定義を与える．次に前節の手法により弱逆関数を導出する．得られた結合的演算子を単純化し, 並列プログラムを導出する．この手続きの中で自動的に行うことができないものは逐次プログラムの定義を与える箇所だけであり, その他のステップは自動的に実行できる．我々はそのうち弱逆関数の導出の実装を行った．単純化と並列プログラムの導出については今後の課題である．

例 6.1. 関数 sum の左方向関数, 右方向関数としての定義は

$$\begin{aligned} sum\ [a] &= a \\ sum\ ([a] +\!\!+\ x) &= a + sum\ x \\ sum\ (x +\!\!+\ [a]) &= sum\ x + a \end{aligned}$$

であるので, sum はリスト準同型の形で定義できることが第三準同型定理よりわかる. 弱逆関数を導出すると $sum^\circ = [\cdot]$ であり, 第三準同型定理と補題 3.2 により

$$\begin{array}{lcl} sum & = & (\!|id,\ \odot|\!) \\ a \odot b & = & sum\ (sum^\circ\ a +\!\!+ sum^\circ\ b) \\ & = & sum\ ([a] +\!\!+ [b]) \end{array}$$

となる. $sum\ ([a] +\!\!+ [b]) = a + b$ と単純化できるので, 得られるリスト準同型の形は

$$\begin{array}{lcl} sum & = & (\!|id,\ \odot|\!) \\ a \odot b & = & a + b \end{array}$$

となる. □

例 6.2. 関数 $(mps \vartriangle sum)$ の左方向関数, 右方向関数としての定義を考えると,

$$\begin{array}{l} (mps \vartriangle sum)\ [a] \\ \quad = \quad (a \uparrow 0,\ a) \\ (mps \vartriangle sum)\ ([a] +\!\!+ x) \\ \quad = \quad (0 \uparrow (a + mps\ x),\ a + sum\ x) \\ (mps \vartriangle sum)\ (x +\!\!+ [a]) \\ \quad = \quad (mps\ x \uparrow (sum\ x + a),\ sum\ x + a) \end{array}$$

となり, 確かに $(mps \vartriangle sum)$ はリスト準同型の形で定義できることがわかる. $(mps \vartriangle sum)$ の弱逆関数を導出すると,

$$\begin{array}{l} (mps \vartriangle sum)^\circ\ (p,\ s) \\ \quad = \textbf{if}\ (p \geq s \wedge 0 \leq p)\ \textbf{then}\ [p,\ s - p] \end{array}$$

となるので, 第三準同型定理と補題 3.2 により

$$\begin{array}{l} (mps \vartriangle sum) = (\!|id,\ \odot|\!) \\ (p_x,\ s_x) \odot (p_y,\ s_y) \\ \quad = (mps \vartriangle sum)\ (f^\circ\ a +\!\!+ f^\circ\ b) \\ \quad = (mps \vartriangle sum)\ ([p_x,\ s_x - p_x] +\!\!+ [p_y,\ s_y - p_y]) \end{array}$$

となる. 単純化すると,

$$\begin{array}{l} (mps \vartriangle sum) = (\!|f,\ \odot|\!) \\ \quad f\ a \qquad\qquad\qquad = (a \uparrow 0,\ a) \\ \quad (x_p, x_s) \odot (y_p, y_s)\ = (x_p \uparrow (x_s + y_p),\ x_s + y_s) \end{array}$$

となり, 例 3.2 で示したプログラムが得られる. これはよく知られた効率的な maximum prefix sum のリスト準同型による定義である. □

例 6.3. 関数 $(mss \vartriangle mps \vartriangle mts \vartriangle sum)$ の左方向関数, 右方向関数の定義は

$$\begin{array}{l} (mss \vartriangle mps \vartriangle mts \vartriangle sum)\ [a] \\ \quad = (a \uparrow 0,\ a \uparrow 0,\ a \uparrow 0,\ a) \\ (mss \vartriangle mps \vartriangle mts \vartriangle sum)\ ([a] +\!\!+ x) \\ \quad = ((a + mps\ x) \uparrow mss\ x,\ 0 \uparrow (a + mps\ x), \\ \qquad mts\ x \uparrow (a + sum\ x),\ a + sum\ x) \\ (mps \vartriangle mps \vartriangle mts \vartriangle sum)\ (x +\!\!+ [a]) \\ \quad = (mss\ x \uparrow (mts\ x + a),\ mps\ x \uparrow (sum\ x + a), \\ \qquad (mts\ x + a) \uparrow 0,\ sum\ x + a) \end{array}$$

と与えられる. 弱逆関数を導出すると,

$$\begin{array}{l} (mss \vartriangle mps \vartriangle mts \vartriangle sum)^\circ\ (m,\ p,\ t,\ s) \\ \quad = \textbf{if}\ (0 \leq p \leq m \wedge 0 \leq t \leq m \wedge s + m \leq t + p) \\ \qquad \textbf{then}\ [p,\ -p - t + s,\ m,\ -m + t] \end{array}$$

となるので, 並列プログラムを導出すると

$$\begin{array}{l} (mss \vartriangle mps \vartriangle mts \vartriangle sum) = (\!|f,\ \odot|\!) \\ f\ a = (a \uparrow 0,\ a \uparrow 0,\ a \uparrow 0,\ a) \\ (m_x,\ p_x,\ t_x,\ s_x) \odot (m_y,\ p_y,\ t_y,\ s_y) \\ \qquad = (m_x \uparrow (t_x + p_y) \uparrow m_y,\ p_x \uparrow (s_x + p_y), \\ \qquad\quad (t_x + s_y) \uparrow t_y,\ s_x + s_y) \end{array}$$

となる. これはよく知られた効率的な maximum segment sum のリスト準同型による定義 [6] である. □

7 関連研究

プログラムの並列化は重要な研究テーマであり, 長い間多くの研究者の注目を集めてきた. 特に結合的な演算の抽出は並列化に関する重要な問題の 1 つである. 結合的な演算の抽出の理論は大きく 2 種類に分けられ, それぞれ第三準同型定理 [12] と context preservation 定理 [5] としてまとめられている. 前者を発展させたものとして, Gorlatch と Geser [11, 15, 16] は第三準同型定理によってリスト準同型の形のプログラムが存在することが分かっている逐次プログラムから, リスト準同型のプログラムを導出する手法を提案した. また後者を発展させ限定された範囲での実装を行った研究として, [10] や [21] がある. これらは本論文の手法とは異なるアプローチを取っており, 我々の手法との比較は今後の課題である.

弱逆関数は逆関数の一般化と捉えることができるが, 逆関数についてはこれまで多くの手法が提案されてきた. 逆関数の導出の考えは [8, 17] までさかのぼり, 自動的な導出として Korf と Eppstein [9, 18] に

よるものがある．実際の実装として [9, 18] を拡張した [13, 14] といったものが挙げられる．弱逆関数と逆関数は密接な関係があり, これらの研究を弱逆関数の導出に利用できると予想される．

8 まとめ

本論文では弱逆関数という概念を導入し, 新しいプログラムの自動的な並列化手法を提案した．弱逆関数は関数の出力から元の入力のうちのひとつを返す関数であり, 弱逆関数を用いることにより第三準同型定理からリスト準同型によって記述された並列プログラムを得ることができる．我々は弱逆関数を自動導出することにより並列プログラムを自動的に得る手法を提案し, 制限を加えた言語上で弱逆関数を導出する手法を示した．さらに実際に弱逆関数の自動導出の効率的な実装を行った．

今後の課題はいくつか考えられる．まず, 今回は弱逆関数の自動導出については実装を行ったが, 並列プログラムの自動導出までは実装していない．我々の手法の有用性を確認するためには実際に並列プログラムを自動導出し, その効率を調査する必要がある．次に, 今回は弱逆関数は常に高々定数長の長さのリストを返すと仮定して導出を行った．しかし, 例えばリストの長さを求める関数 *length* のように, 弱逆関数の出力するリストの長さが高々定数とならない, しかし並列化に有用な関数もある．このような関数に対しても弱逆関数と第三準同型定理を利用できるような枠組みを考えたい．

謝辞

本研究についての議論に積極的に参加し，多くの着想の基を与えてくださった篠埜功さんと Shin-Cheng Mu さんに感謝します．

参考文献

[1] The Omega project. `http://www.cs.umd.edu/projects/omega/`.

[2] J. Bentley. Algorithm design techniques. In *Programming Pearls,* Column 7, pp. 69–80. Addison-Wesley, 1986.

[3] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, Vol. 36 of *NATO ASI Series F*, pp. 5–42. Springer-Verlag, 1987.

[4] R. S. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[5] W. N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. In *IEEE Computer Society International Conference on Computer Languages (ICCL'98)*, pp. 153–162. IEEE Press, 1998.

[6] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problems. Report CSR-25-93, Department of Computing Science, The University of Edinburgh, 1993.

[7] D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, Vol. 7, pp. 91–99, 1972.

[8] E. W. Dijkstra. Program inversion. In Friedrich L. Bauer and Manfred Broy, editors, *Program Construction*, Vol. 69 of *Lecture Notes in Computer Science*, pp. 54–57. Springer, 1978.

[9] D. Eppstein. A heuristic approach to program inversion. In *Proceedings of the 9th International Joint Conferences on Artificial Intelligence*, pp. 219–221, Los Angeles, CA, USA, 1985.

[10] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, pp. 135–146, 1994.

[11] A. Geser and S. Gorlatch. Parallelizing functional programs by generalization. In M. Hanus, J. Heering, and K. Meinke, editors, *Algebraic and Logic Programming (ALP'97)*, Vol. 1298 of *Lecture Notes in Computer Science*, pp. 46–60. Springer-Verlag, 1997.

[12] J. Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, Vol. 6, No. 4, pp. 657–665, 1996.

[13] R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori, editor, *Programming Languages and Systems. Proceedings*, Vol. 2895 of *Lecture Notes in Computer Science*, pp. 246–264. Springer-Verlag, 2003.

[14] R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on LR parsing. In Y. Kaneyama and P. J. Stuckey, editors, *Functional and Logic Programming. Proceedings*, Vol. 2998 of *Lecture Notes in Computer Science*, pp. 291–306. Springer-Verlag, 2004.

[15] S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[16] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In H. Kuchen and D. Swierstra, editors, *Programming languages: Implementation, Logics and Programs. PLILP'96*, Lecture Notes in Computer Science 1140, pp. 274–288. Springer-Verlag, 1996.

[17] D. Gries. Inverting programs. In *The Science of Programming*, chapter 21, pp. 265–274. Springer-Verlag, 1981.

[18] R. E. Korf. Inversion of applicative programs. In *Proceedings of the 7th International Conferences on Artificial Intelligence*, pp. 1007–1009, Vancouver, Canada, 1981.

[19] M. Pressburger. Uber die vollstandigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervorstritt. *Sprawozdanie z I Kongresu Matematikow Krajow Slowcanskich Warszawa*, pp. 92–101, 1929.

[20] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedisgs of the 1991 ACM/IEEE conference on Supercomputing*, pp. 4–13, Albuquerque, NM, USA, 1991.

[21] D. N. Xu, S. C. Khoo, and Z. Hu. PType system: A featherweight parallelizability detector. In W.-N. Chin, editor, *Proceedings of Second Asian Symposium on Programming Languages and Systems (APLAS 2004)*, Vol. 3302 of *Lecture Note in Computer Science*, pp. 197–212. Springer-Verlag, 2004.