情報処理学会論文誌　プログラミング　Vol. 2　No. 2　132–143 (Mar. 2009)

並列プログラムの候補生成と適合性検査による並列化

森畑明昌[†1]　松崎公紀[†1]
胡振江[†2]　武市正人[†1]

近年，並列計算のための環境は身近になってきている．しかし，効率の良い並列プログラムの構築は逐次プログラムの構築に比べてはるかに難しい．そのため，逐次プログラムをもとにして自動的に並列プログラムを得る**自動並列化**の手法が求められている．プログラムの並列化に関連して，関数プログラミングの分野では**第三準同型定理**という定理が知られている．第三準同型定理は，配列からある値を計算する問題に対し，その配列の要素を右から順に走査するプログラムと左から順に走査するプログラムの両方が存在すれば，その問題を分割統治法によって解く並列プログラムが存在する，ということを示している．第三準同型定理は並列プログラムの構成に有用であり，第三準同型定理に基づいた自動並列化手法もいくつか提案されている．本論文では第三準同型定理に基づいた新たな自動並列化手法を提案する．提案手法では，逐次プログラムを元に並列プログラムの候補を生成し，それらの中から正しい並列プログラムとなっているものを選択する．さらに，我々は提案手法に基づいた自動並列化器を試作した．我々の並列化器は，算術演算と条件式によって定義された再帰関数から，その再帰関数を計算する並列プログラムを自動的に生成することができる．本論文では我々の自動並列化手法の概要について述べた後，我々の実装と実験結果について報告する．

Program Parallelization by Candidate Generation and Conformity Testing

Akimasa Morihata,[†1] Kiminori Matsuzaki,[†1]
Zhenjiang Hu[†2] and Masato Takeichi[†1]

Recently, it has been easy to access parallel computation environments. However, efficient parallel programs are much harder to develop than sequential ones. Therefore, *automatic parallelization methods*, which generate parallel programs from sequential ones, are called for. *The third homomorphism theorem* is a folk theorem in the functional programming community. The theorem states that for a problem to compute a value from an array, there exists a divide-and-conquer parallel algorithm to solve the problem if and only if the problem can be solved by both of two programs that respectively scan the array in leftward and rightward manners. The theorem is useful for developing parallel programs, and automatic parallelization methods have been proposed on it. In this paper, we propose a new automatic parallelization method. We generate candidates of parallel programs, and choose one that satisfies the requirement of parallel programs. We implemented our idea as an automatic parallelization system, which can parallelize recursive functions defined by arithmetic and conditional expressions.

1. プログラムの自動並列化

近年，並列計算のための環境は身近になってきている．PC クラスタは容易に利用できるようになり，マルチプロセッサ・マルチコア PC も一般に出回るようになってきた．しかし，効率の良い並列プログラムの構築は逐次プログラムの構築に比べてはるかに難しく，専門家以外にとっては身近になってきた並列計算環境を有効活用しにくかった．そのため，逐次プログラムをもとにして自動的に並列プログラムを得る**自動並列化**の手法が求められている．

本論文で議論するのは配列を走査するループの自動並列化である．例として図 1 から図 4 までの 4 つのプログラムを考えてみる．**図 1** は配列の各要素を自乗するプログラムである．このプログラムのように，配列のすべての要素に独立な計算を行う場合，並列化は容易である．配列の添え字ごとに担当するプロセッサを割り当てればよい．**図 2** は配列中の要素の中で最大のものを求めるプログラムである．最大値を並列計算するためには，各プロセッサにそれぞれ配列の一部分を割り当て，各々のプロセッサがそれぞれの担当分の最大値を発見し，それらの中で最大のものを求める，という手順が考えられる．しかし，自動並列化を行うためには，図 2 のプログラムが最大値を計算すること，前述の手順が確かに最大値を求めること，の 2 点を自動的に判定しなければならない．**図 3** は，先頭要素を含む連続した部分列の要素和の中で最大のものを求めるプログラムである．また，**図 4** は，文字列から数値への変換を行うプログラムである．このようなプログラムに対しては，ループ中の計算が複雑であるため，人手による並列アルゴリズムの考案すら容易ではない．

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 国立情報学研究所
National Institute of Informatics

```
for (i = 0; i < N; ++i) {
    a[i] = a[i] * a[i];
}
```

図 1　各要素を自乗するプログラム
Fig. 1　Computing squares of elements.

```
int m = 0;
for (i = 0; i < N; ++i) {
    if (m < a[i]) m = a[i];
}
```

図 2　最大値を求めるプログラム
Fig. 2　Finding the maximum number.

```
int m = 0;
for (i = N-1; i >=0; --i) {
    if (m >= 0) m += a[i];
    else m = a[i];
}
```

図 3　最大接頭部和を計算するプログラム
Fig. 3　Computing the maximum prefix sum.

```
int s = 0;
for (i = 0; i < N; ++i) {
    s = s * 10 + (a[i] - '0');
}
```

図 4　文字列から数値への変換を行うプログラム
Fig. 4　Converting a string to a number.

プログラムの並列化に関連して，関数プログラミングの分野では**第三準同型定理**[11]という定理が知られている．第三準同型定理は，配列からある値を計算する問題に対し，その配列の要素を右から順に走査するプログラムと左から順に走査するプログラムの両方が存在すれば，その問題を分割統治法によって解く並列プログラムが存在する，ということを示している．第三準同型定理は，左右それぞれから配列を走査する 2 種類のプログラムを記述することで並列化に有用な情報を発見することができる，ということを示唆する．実際，第三準同型定理に基づき，2 つの逐次プログラムから並列プログラムを得る自動並列化手法がいくつか提案されている[10),16),22)]．

本論文では第三準同型定理に基づいた新たな自動並列化手法を提案する．提案手法の特徴は，並列プログラムの候補生成と適合性検査に基づいている，という点である．すなわち，まず逐次プログラムをもとに並列プログラムの候補を生成し，その後，候補の中から正しい並列プログラムを発見する．提案手法に基づき，我々は自動並列化器を試作した．我々の並列化器では再帰関数を用いて記述された逐次プログラムから並列プログラムを自動的に生成する．再帰関数の記述には算術演算および条件式を用いることができる．また，並列プログラムの正しさの検証には**限定記号除去**[5),6),14)]（quantifier elimination）を利用する．

本論文では，まず最初に第三準同型定理を概観する（2 章）．次に我々の自動並列化手法の概要について述べ（3 章），我々の実装（4 章）および実験結果（5 章）について報告する．最後に関連研究と今後の課題を議論して（6 章）まとめに代える．

2. 第三準同型定理

本論文を通して以下の記法を用いる．二項演算子 $\uparrow$ と $\downarrow$ はそれぞれ 2 値の最大値および最小値を表す．配列は角括弧で囲まれカンマで区切られた値の列で表現する．たとえば，$[80, 65, 82, 65]$ は長さ 4 の配列であり，その最初の要素は 80，2 番目と 4 番目の要素は 65，3 番目の要素は 82 である．二項演算子 $+\!\!+$ は配列の連結を表現する．たとえば，$[80, 65, 82, 65] +\!\!+ [76, 69, 76, 76] = [80, 65, 82, 65, 76, 69, 76, 76]$ である．また，本論文では空配列は考えない．

2.1　右方向関数，左方向関数，リスト準同型

本論文では配列を走査する計算を再帰関数を用いて表現する．

例として配列の要素和の計算を考える．配列の要素和の計算は以下の 2 つの等式によって定義される関数 sum によって表現できる．

$$sum([a]) = a$$
$$sum([a] +\!\!+ x) = a + sum(x)$$

1 つ目の等式は 1 要素の配列に対する計算を，2 つ目の等式は複数要素を持つ配列に対する計算を表現している．2 つ目の等式は，配列の左端以外の要素の和 $sum(x)$ から配列全体の要素和 $sum([a] +\!\!+ x)$ を求められることを示している．そのため，この等式は配列の要素和を右から左に求める計算に対応する．

配列の要素和は左から右に求めることもできる．

$$sum([a]) = a$$

$sum(x +\!\!+ [a]) = sum(x) + a$

上の等式は，複数要素を持つ配列に対し，その和が右端以外の要素の和を用いて計算できることを示している．

今まで見てきた 2 種類の等式群はいずれも逐次プログラムに対応する．一方，以下の等式は並列プログラムと見ることができるものである．

$sum([a]) \quad = a$

$sum(x +\!\!+ y) = sum(x) + sum(y)$

2 つ目の等式の意味は，任意の配列 x と y に対して，x と y の連結 $x +\!\!+ y$ の要素和は配列 x の要素和と配列 y の要素和の和である，ということである．換言すると，配列の要素和はその配列を 2 つの部分列に分割しそれぞれの要素和の和を求めることで計算できる．このとき，各部分列の要素和は独立な計算であるため，複数のプロセッサにより同時に行うことができる．そのため並列プログラムと見なすことができる．

それでは今までの議論を定式化する．

定義 1（左方向関数） 関数 f が左方向関数であるとは，ある演算子 $\ominus$ が存在して $f([a] +\!\!+ x) = a \ominus f(x)$ を満たすことである． □

定義 2（右方向関数） 関数 f が右方向関数であるとは，ある演算子 $\oslash$ が存在して $f(x +\!\!+ [a]) = f(x) \oslash a$ を満たすことである． □

定義 3（リスト準同型[2]） 関数 f がリスト準同型であるとは，ある演算子 $\odot$ が存在して $f(x +\!\!+ y) = f(x) \odot f(y)$ を満たすことである． □

要素和の計算 sum に関して先ほど 3 種類の等式群を示した．これらは順に，sum が左方向関数であること，sum が右方向関数であること，そして sum がリスト準同型であること，を示している．

ある関数がリスト準同型であれば，その関数は分割統治法により並列計算できる．

補題 4 関数 f が $f(x +\!\!+ y) = f(x) \odot f(y)$ を満たすとき，p プロセッサからなる排他読み込み・排他書き込み並列ランダムアクセス機械（exclusive-read/exclusive-write parallel random access machine）上で，f は長さ n の配列に対し $T_f(n/p) + T_{\odot} \log p$ 時間で評価できる．ただし，$T_f(k)$ は f の長さ k の配列に対する逐次での評価に要する時間，$T_{\odot}$ は $\odot$ の評価に要する時間である． □

補題 4 は，リスト準同型が効率の良い並列プログラムとなりうることを示している．いま，$T_f(k)$ が k に比例するとし，$T_{\odot}$ を $\mathrm{O}(1)$，計算対象の配列の長さを n とする．このとき，補題 4 はプロセッサ数に対して線形の台数効果が $\mathrm{O}(n/\log n)$ 台まで見込めることを示唆する．すなわち，リスト準同型に基づく並列プログラムは，多量のプロセッサを用いることで非常に高速な計算が期待できる．本論文の目的は，左方向関数や右方向関数としての関数定義から，リスト準同型としての関数定義を得ることである．

2.2　第三準同型定理

第三準同型定理[11] は関数型プログラミングの分野で知られている定理である．この定理は，ある関数がリスト準同型であるための必要十分条件を示している．

定理 5（第三準同型定理[11]） 関数 f が右方向関数かつ左方向関数であるとき，またそのときに限り，f はリスト準同型である． □

並列プログラムは逐次プログラムに比べて複雑である．そのため，1 つの逐次プログラムから 1 つの並列プログラムを得るのは，一般にはほとんど絶望的である．しかし，第三準同型定理は，左方向と右方向の 2 つの逐次プログラムがあれば 1 つの並列プログラムを得るのに十分な情報を含んでいる，ということを示唆している．この点が第三準同型定理の興味深い部分である．

一方で，第三準同型定理は，ある関数 f がリスト準同型であるための必要十分条件を与えてはいるが，具体的に $f(x +\!\!+ y) = f(x) \odot f(y)$ を満たす演算子 $\odot$ を得る方法は示していない．そのため，第三準同型定理のみからは並列プログラムは得られない．

本論文で提案する並列化手法は以下に示す第三準同型定理の系（系 7）に基づいている．証明は第三準同型定理と組化変換[4),13)] から直截である．

定義 6（両方向関数集合） 関数集合 $\{f_1, \ldots, f_n\}$ が両方向であるとは，すべての i（$1 \leq i \leq n$）に対し $f_i([a] +\!\!+ x) = \lambda_i(a, f_1(x), f_2(x), \ldots, f_n(x))$ および $f_i(x +\!\!+ [a]) = \rho_i(f_1(x), f_2(x), \ldots, f_n(x), a)$ を満たす関数 λ_i および ρ_i とが存在することである． □

系 7 関数集合 $\{f_1, f_2, \ldots, f_n\}$ が両方向であるとき，またそのときに限り，関数 $g(x) = (f_1(x), f_2(x), \ldots, f_n(x))$ はリスト準同型である． □

2.3　例

配列の最大要素

配列中の最大要素を求める関数 max を考える．この関数は右方向かつ左方向である．

$max([a]) \quad = a$

$max([a] +\!\!+ x) = \mathsf{if}\ a > max(x)\ \mathsf{then}\ a\ \mathsf{else}\ max(x)$

$max(x +\!\!+ [a]) = \mathsf{if}\ a > max(x)\ \mathsf{then}\ a\ \mathsf{else}\ max(x)$

よって，第三準同型定理より，関数 max はリスト準同型である．

文字列の数値変換

次に文字列を数値に変換する関数 $atoi$ を考える．関数 $atoi$ は右方向である．

$$atoi([a]) = a - \text{‘0’}$$
$$atoi(x +\!\!+ [a]) = atoi(x) \times 10 + (a - \text{‘0’})$$

しかし，右から左へ配列を走査しながら文字列を数値に変換するには，以下で定義される補助関数 $pow10$ が必要である．

$$atoi([a] +\!\!+ x) = (a - \text{‘0’}) \times pow10(x) + atoi(x)$$
$$pow10([a]) = 10$$
$$pow10([a] +\!\!+ x) = 10 \times pow10(x)$$
$$pow10(x +\!\!+ [a]) = pow10(x) \times 10$$

よって $atoi$ は左方向ではない．しかし，上の等式群は関数集合 $\{atoi, pow10\}$ が両方向であることも示している．よって，系 7 より，2 関数 $atoi$ および $pow10$ を同時に計算するのであれば，その計算はリスト準同型によって表現できる．

関数 $atoi$ の並列化に実用的な意義はほとんどない．しかし，$atoi$ の並列化は一般的な多項式の計算の並列化につながる．実際，ある配列 $[a_0, a_1, \ldots, a_n]$ と定数 b に対し，$f(a_0) \times b^0 + f(a_1) \times b^1 + \cdots + f(a_n) \times b^n$ を求める計算は無限級数展開をはじめとして実用上頻繁に登場する．関数 $atoi$ はこのような計算の一例にあたり，具体的には $f(a) = a - \text{‘0’}$, $b = 10$ である．

最大接頭部分列和

最大接頭部和問題[16] とは，ある配列に対し，その先頭要素を含む連続した部分列の要素和の中で最大のものを求める問題である．たとえば，最大接頭部和問題を解く関数を mps とすると，$mps([-6, 9, 1, -4, 6, -7, 4]) = (-6) + 9 + 1 + (-4) + 6 = 6$ である．この問題は，簡単ではあるが，配列からの知識発見の一例となっている．

関数 mps は左方向だが右方向でない．しかし，関数集合 $\{mps, sum\}$ は両方向である．

$$mps([a]) = a$$
$$mps([a] +\!\!+ x) = a \uparrow (a + mps(x))$$
$$mps(x +\!\!+ [a]) = mps(x) \uparrow (sum(x) + a)$$

関数 sum が右方向かつ左方向であることは前に示した．よって，系 7 より，2 関数 mps および sum を同時に計算するのであれば，その計算はリスト準同型によって表現できる．

3. 並列プログラムの候補生成と適合性検査

本論文で提案する手法では，まず並列プログラムの候補を生成し，その中からリスト準同型であるものを発見することで並列プログラムを得る．具体的には，入力された関数 f に対し，以下の 2 手順によってリスト準同型の定義を得ることを試みる．

(1)　$f([a] +\!\!+ y) = f([a]) \ominus f(y)$ を満たす演算子 $\ominus$ を列挙する．

(2)　任意の配列 x および y について $f(x +\!\!+ y) = f(x) \ominus f(y)$ が成り立つことを証明する．

しかし，単純に上の処理を行うだけでは多くの場合並列化は失敗する．なぜなら，$f(x +\!\!+ y) = f(x) \ominus f(y)$ を満たす演算子 $\ominus$ はそもそも存在しないかもしれないからである．そのため，我々は右方向と左方向の 2 つの関数定義をユーザに要求する．これにより，所望の演算子の存在は第三準同型定理から保証される．そのうえ，両方向の関数定義に含まれる情報により効果的に並列化が行えることも期待できる．

以下，提案手法について具体的に説明する．例としては，2.3 節で示した，最大接頭部和問題を解く関数 mps を用いる．

3.1 入　　力

入力は両方向関数集合およびそれらの定義である．たとえば mps であれば mps の定義だけではなく sum の定義も必要である．また，sum はそれのみで両方向関数集合となり，mps は sum と合わせることで両方向関数集合となる．これらの情報もユーザから与えられることを仮定する．2.3 節で見たとおり，これらの情報は，mps と sum に対する左右両方向の定義を書き下すことで得られる．

3.2 候補生成と適合性検査

入力された両方向関数集合の部分集合で両方向関数集合をなすものすべてに対し，部分集合関係で小さいものから順に以下の「候補生成」と「適合性検査」を繰り返す．入力された両方向関数集合全体を処理した後，「出力」を行う．たとえば mps であれば，まず $\{sum\}$ について候補生成と適合性検査を行い，次に $\{mps, sum\}$ について処理し，最後に出力を行う．

この部分の処理では，各関数は「処理済み」または「未処理」に分類される．初期状態ではすべての関数は「未処理」である．各関数（たとえば f とする）は，並列化が完了すると「処理済み」となり，並列化の結果である式 e_f^* を割り当てられる．

3.2.1 候 補 生 成

両方向関数集合 $\{f_1, f_2, \ldots, f_k\}$ 中の各関数 f_i について，f_i が処理済みなら e_{f_i} として

$e^*_{f_i}$ を用い，未処理なら

$$f_i([a] +\!\!+ y) = e_{f_i}[f_1([a])/L_{f_1}, \ldots, f_k([a])/L_{f_k}, f_1(y)/R_{f_1}, \ldots, f_k(y)/R_{f_k}]$$

を満たし変数 a および y を含まない式 e_{f_i} を生成する．ここで $L_{f_1}, \ldots, L_{f_k}, R_{f_1}, \ldots, R_{f_k}$ は新しい変数である．こうして得られた式の集合 $\{e_{f_1}, e_{f_2}, \ldots, e_{f_k}\}$ が 1 つの候補であり，次の「適合性検査」の入力となる．

例として，$\{mps, sum\}$ についての処理を考える．この時点で sum は処理済みであり，$e^*_{sum} = L_{sum} + R_{sum}$ が得られている．mps は未処理なので，上の条件を満たす e_{mps} を生成する．たとえば，

$$mps([a] +\!\!+ x) = mps([a]) \uparrow (sum([a]) + mps(x))$$

が成り立つため，以下の e_{mps} は所望の条件を満たす．

$$e_{mps} = L_{mps} \uparrow (L_{sum} + R_{mps})$$

よって，$\{e_{mps}, e^*_{sum}\}$ が 1 つの候補となる．

3.2.2　適合性検査

候補 $\{e_{f_1}, e_{f_2}, \ldots, e_{f_k}\}$ に対し，

$$\forall e_{f_i} \in \{e_{f_1}, e_{f_2}, \ldots, e_{f_k}\}:$$
$$f_i(x +\!\!+ y) = e_{f_i}[f_1(x)/L_{f_1}, \ldots, f_k(x)/L_{f_k}, f_1(y)/R_{f_1}, \ldots, f_k(y)/R_{f_k}]$$

の証明を試みる．ただし関数 f_i が処理済みである場合，e_{f_i} に関する証明は省略して問題ない．

証明が成功した場合，すべての処理済みでない関数 f_i を処理済みとし，$e^*_{f_i}$ を e_{f_i} とする．証明が失敗した場合，「候補生成」に戻り新たな候補の生成を試みる．考えうるすべての候補について証明が失敗した場合，並列化は失敗である．

たとえば，前述の候補 $\{e_{mps}, e^*_{sum}\}$ に対する証明は成功する．この場合 mps は処理済みとなり e^*_{mps} として e_{mps} が記憶される．

3.3　出　　力

並列化が失敗した場合，その旨を報告する．そうでなければ，以下の演算子 $\odot$ を出力する．ただし，以下では入力された関数集合が $\{f_1, f_2, \ldots, f_k\}$ であるとしている．

$$(L_{f_1}, \ldots, L_{f_k}) \odot (R_{f_1}, \ldots, R_{f_k}) = (e^*_{f_1}, \ldots, e^*_{f_k})$$

系 7 より，関数 $g(x) = (f_1(x), f_2(x), \ldots, f_k(x))$ はリスト準同型である．かつ，出力された演算子 $\odot$ は $g(x +\!\!+ y) = g(x) \odot g(y)$ を満たすことが適合性検査により証明されている．

最大接頭部和を解く関数 mps に対しては，$e^*_{sum} = L_{sum} + R_{sum}$ および $e^*_{mps} = L_{mps} \uparrow (L_{sum} + R_{mps})$ が得られる．よって，出力される演算子は以下である．

$$(L_{mps}, L_{sum}) \odot (R_{mps}, R_{sum}) = (L_{mps} \uparrow (L_{sum} + R_{mps}), L_{sum} + R_{sum})$$

3.4　特　　徴

本手法の最大の特徴は候補生成と適合性検査に基づいている点である．そのため，実装の際には，候補生成および適合性検査をいかに行うかが問題となる．一方で，目的や対象とするプログラムごとに候補生成および適合性検査の方法を変化させることで，効果的な並列化手法を構築できることが期待できる．

候補生成の戦略としては，たとえば効率良く計算できる式のみを候補として生成するという方法が考えられる．これにより，非効率な並列プログラムを得ることを避けることができる．また，他の並列化手法で得られたものを候補と見なす方法も考えられる．この場合，我々の手法は，いくつかの並列化手法を組み合わせ，それらの結果を検証する手法だと見ることができる．

適合性検査では，既知の自動定理証明の手法が利用できる．自動定理証明はさかんに研究されている分野であり，並列化したいプログラムごとに適切な手法を選ぶことで，多くのプログラムを実用的な時間で並列化できることが期待できる．

4.　自動並列化器の実装

我々は前章で提案した手法に従い自動並列化器を試作した．実装は関数型言語 Haskell[17] によって行われている．

4.1　入 力 言 語

図 5 に 並列化器への入力プログラムを記述するための言語の構文を示す．プログラムは関数定義の列である．各関数は左方向の形で記述し，右方向の関数定義の代わりに，各関数がどの関数とまとめることで両方向関数集合となるかを付記する．すべての関数は整数値の

$prog$	::=	$decl \cdots decl$	{ プログラム }
$decl$	::=	$f \leftarrow (f, \ldots, f);\ f([a]) = e;\ f([a] +\!\!+ x) = e;$	{ 関数定義 }
e	::=	$c \mid a \mid f(x) \mid e + e \mid e - e \mid e \times e \mid e \uparrow e \mid e \downarrow e$	
		$\mid$ if p then e else e	{ 式 }
p	::=	$p \wedge p \mid p \vee p \mid \neg p \mid e < e \mid e = e \mid e \leq e \mid e \neq e$	{ 述語 }

図 5　並列化器への入力プログラムを記述する言語の構文（c は整数定数，a は整数値変数）

Fig. 5　The syntax of the language for describing inputs of our system, where c denotes a constant integer and a denotes an integer-valued variable.

```
mps <- (sum);
mps([a]) = a;
mps([a]++x) = max(a, a + mps(x));
sum <- ();
sum([a]) = a;
sum([a]++x) = a + sum(x);
```

図 6　並列化器への入力プログラム例（最大接頭部和問題を解く関数）
Fig. 6　An input program for our automatic parallelizer, where the function mps solves the maximum prefix sum problem.

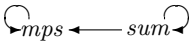

図 7　並列化のための依存関係グラフ（関数 mps と sum）
Fig. 7　The dependency graph of the functions mps and sum.

配列を走査し整数値を計算する．そのため，真偽値や文字も整数値で表現する必要がある．値の計算には，定数，配列の要素の値，算術演算，再帰関数の呼び出し，そして条件式を用いることができる．条件式の述語は等式と不等式の組合せによって記述される．

例として，最大接頭部和問題を解く関数 mps のこの言語による記述を**図 6** に示す．2.3 節で示したとおり，関数 mps は配列の要素和を計算する関数 sum と合わせることで両方向関数集合となる．プログラムの 1 行目 `mps <- (sum);` はこのことを表現している．また sum はそれ単独で両方向であり，プログラムの 4 行目 `sum <- ();` はこのことを表現している．

4.2　並列化アルゴリズムの実装

4.2.1　前　処　理

まず入力された関数群から両方向関数集合を求める．このためには，各関数を頂点，「関数 f は両方向関数集合となるには関数 g が必要」という関係を g から f への有向辺，と見なすことで得られるグラフの強連結成分をトポロジカルソートすればよい．

関数 mps と関数 sum の場合のグラフを**図 7** に示す．各関数は自分自身に対する辺を持つ．また，mps は両方向関数集合となるには sum が必要であるため，sum から mps へ向かう有向辺がある．このグラフでは $\{mps\}$ および $\{sum\}$ が強連結成分であり，トポロジカルソートにより $[\{sum\},\{mps\}]$ という順序が得られる．よって，まず $\{sum\}$ を並列化し，その後 $\{sum, mps\}$ を並列化する．

4.2.2　候補の生成

候補の生成には，配列の長さが 2 以上の場合に対する定義の右辺式に対する，配列の長さが 1 の場合の定義の右辺式の非決定的な代入を用いる．

以下，関数集合 $\{sum, mps\}$ に対する候補生成を例として説明する．関数 sum と mps とは，ともに長さ 1 の配列 $[a]$ に対して配列要素の値 a を返す．よって，等式 $mps([a] +\!\!+ x) = a \uparrow (a + mps(x))$ の右辺式の部分式 a を $mps([a])$ または $sum([a])$ で置換しても等式は成り立つ．よって，2 カ所の a それぞれについて，「$mps([a])$ に置換する場合」「$sum([a])$ に置換する場合」「置換しない場合」を考える．ただし，最終的に変数 a が残るものは不要であるため，今回は「置換しない場合」は破棄される．結果，以下の 4 式が得られる．

$$mps([a] +\!\!+ x) = mps([a]) \uparrow (mps([a]) + mps(x))$$
$$mps([a] +\!\!+ x) = mps([a]) \uparrow (sum([a]) + mps(x))$$
$$mps([a] +\!\!+ x) = sum([a]) \uparrow (mps([a]) + mps(x))$$
$$mps([a] +\!\!+ x) = sum([a]) \uparrow (sum([a]) + mps(x))$$

上の 4 式のそれぞれの右辺式に対し，$mps([a])$ を L_{mps} で，$sum([a])$ を L_{sum} で，$mps(x)$ を R_{mps} で，（今回は現れなかったが）$sum(x)$ を R_{sum} で，それぞれ置き換えたものが所望の式である．

本実装では，上記の手続きで得られる候補すべてを考え，そのそれぞれについて順次並列に適合性検査を行い，適合性検査に成功した候補が発見できた時点で停止する．なお，候補の生成順に特別な戦略は用いていない．

4.2.3　適合性検査

適合性検査は配列の長さに対する帰納法と**限定記号除去**[5),6),14)]（quantifier elimination）とを用いて行う．

以下，例として 3.2.1 項で示した候補 $\{e_{mps}, e^*_{sum}\}$ を考える．いま，未処理な関数は mps のみであり，

$$mps(x +\!\!+ y) = mps(x) \uparrow (sum(x) + mps(y))$$

を証明するのが目的である．

まず x の長さが 1 の場合を考える．この場合は候補生成のアルゴリズムより明らかに成り立つ．次に x の長さが 1 以上の場合，すなわち $x = [a] +\!\!+ z$ の場合を考える．以下，等式の両辺を変形する．

$$mps([a] +\!\!+ z +\!\!+ y) = mps([a] +\!\!+ z) \uparrow (sum([a] +\!\!+ z) + mps(y))$$
$$\Leftrightarrow \{\ mps \text{ および } sum \text{ の定義}\ \}$$
$$a \uparrow (a + mps(z +\!\!+ y)) = (a \uparrow (a + mps(z))) \uparrow ((a + sum(z)) + mps(y))$$
$$\Leftrightarrow \{\ \text{帰納法の仮定}\ \}$$

$$a \uparrow (a + (mps(z) \uparrow (sum(z) + mps(y))))$$
$$= (a \uparrow (a + mps(z))) \uparrow ((a + sum(z)) + mps(y))$$

以上より，任意の a，z，y について上式が成り立つことを証明すればよい．このためには，$mps(z)$，$sum(z)$，$mps(y)$ を変数と見なすことで得られる以下の式の証明で十分である．

$$\forall a, L_{mps}, L_{sum}, R_{mps}:$$
$$a \uparrow (a + (L_{mps} \uparrow (L_{sum} + R_{mps}))) = (a \uparrow (a + L_{mps})) \uparrow ((a + L_{sum}) + R_{mps})$$

この式は全称限定記号付きの不等式と見なすことができ，限定記号除去によって証明を行うことができる．

我々は比較のために以下の 2 種類の限定記号除去アルゴリズムを実装した．

1 つは Fourier-Motzkin elimination[6] である．このアルゴリズムは，全称限定記号の除去のたびに式を乗法標準形（conjunctive normal form）に変換する必要があるため，一般には効率が悪い．そのため，式中に変数どうしの積がないことを仮定して実装を行った．この仮定の下では，全称限定記号除去の際に乗法標準形が崩れることがないため，効率の良い実装となっている．

もう 1 つは Loos ら[14] の手法である．これは Fourier-Motzkin elimination よりも一般には効率が良く，変数どうしの積も自乗でなければ場合分けによって扱うことができる．また，効率の向上には式の単純化が非常に有効である[8]．我々は式の平坦化，共通式の削除，値の確定した式の評価，以上 3 つの単純化を実装した．

4.2.4 出　　力

並列化器は C++のプログラムを出力する．出力は並列計算に用いる関数オブジェクトとそのためのデータ構造の定義からなり，並列計算ライブラリ SkeTo[15] から利用できる．

図 8 は並列化器に図 6 のプログラムを入力して得られる出力である．関数オブジェクト `func` は長さ 1 の配列に対する計算に，関数オブジェクト `odot` は部分配列の結果を併合する計算に用いられる．並列計算のためには *mps* の結果と *sum* の結果の両方が必要である．そのため，計算結果を格納する構造体 `my_tuple_t` が定義されている．

4.3 特　　徴

本実装の長所の 1 つは生成される並列プログラムの効率である．候補生成のアルゴリズムより，生成される候補の計算コストは必ず入力された関数群の計算コストの和よりも小さくなる．そのため，効率の悪い並列プログラムが得られることを危惧する必要がない．

一方で，本実装による並列化の成否はユーザによる関数記述に大きく依存する．なぜなら，候補生成のアルゴリズムが関数記述を構文的にのみ処理するため，適切な候補が生成されるか否かはユーザの記述次第であるからである．この点については後に 6.2 節で詳しく議論する．

```
struct my_tuple_t {
  int mis, sum;
  my_tuple_t(int mis, int sum) : mis(mis), sum(sum){}
  my_tuple_t(){}
};

struct func_t : public sketo::functions::base<my_tuple_t(int)> {
  my_tuple_t operator()(const int a) const {
    const int mis = a;
    const int sum = a;
    return my_tuple_t(mis, sum);
  }
} func;

struct odot_t : public sketo::functions::base<my_tuple_t(my_tuple_t,my_tuple_t)> {
  my_tuple_t operator()(const my_tuple_t &x, const my_tuple_t &y) const {
    const int mis = std::max(x.mis,(x.sum+y.mis));
    const int sum = (x.sum+y.sum);
    return my_tuple_t(mis, sum);
  }
} odot;
```

図 8　並列化器に図 6 のプログラムを入力して得られる出力
Fig. 8　The output of our system obtained by inputting the program in Fig. 6.

5. 実　　験

我々の手法の有効性を検証するため，いくつかの実験を行った．例としては，配列の長さの計算（*length*），配列の要素の最大値の計算（*max*），最大接頭部和問題（*mps*），最大部分列和問題[1),16)]（*mss*），文字列から数値への変換（*atoi*），以上 6 つを用いた．なお，最大部分列和問題は連続した部分列の要素和で最大のものを求める問題である．また，関数 *atoi* へ入力文字列は整数型の配列で与えた．

5.1 並列化器の評価実験

我々の候補生成アルゴリズムによって生成される候補の数を**表 1** に示す．生成される候補は全体的にかなり少ないことが分かる．

表 2 に我々の 2 種類の実装および我々が以前に提案した手法[16),22)]（以下 Morita らの手

表 1　各問題で生成する候補数
Table 1　The number of candidates generated for each problem.

	length	*max*	*mps*	*mss*	*atoi*
候補数	2	1	5	24	3

表 2　各並列化器の扱える問題の範囲と実行時間の比較（秒）
Table 2　Comparison of parallelization systems by their feasibility and speed (unit: second)

	length	*max*	*mps*	*mss*	*atoi*
Fourier-Motzkin elimination [6] による実装	0.01	0.01	0.01	0.04	不可
Loos ら[14] の手法による実装	0.01	0.01	0.01	5.25	0.01
Morita らの手法[16),22)]	不可	可	可	可	不可

法と呼称）の比較を示した．実験に用いた環境は CPU が Intel Core2 Duo 1.8 GHz，メモリ 1.5 GB，コンパイラは GHC 6.8.2，OS は Windows XP SP3 である．Morita らの実装[16),22)] については各問題が扱えるか否かのみを記している．また，実行時間にはファイルの入出力の時間を含んでいる．

両方の実装について，多くの例では即座に並列プログラムを出力している．最も長い時間を要した例は最大部分列和問題（*mss*）である．この例では，狭い範囲の問題に特化した実装（Fourier-Motzkin elimination による実装）では短時間で並列化が完了しているのに対し，より一般的な手法による実装（Loos らの手法による実装）では多少時間がかかっている．これから，対象に適した手法によって適合性検査を行うことが重要であることが分かる．

また，配列の長さの計算（*length*）や文字列から数値への変換（*atoi*）は，Morita らの手法では扱うことができなかったが今回の並列化器では並列化できる．この点は本手法の長所である．一方で，Morita らの手法では扱うことができるが，今回の実装では並列化器では並列化できない例もある．これについては 6.2 節で議論する．

5.2　並列化器の生成したプログラムの評価実験

並列化器が生成した並列プログラムについて，長さ 2^{27} の配列に対して計算を行い，その実行時間を計測した．実装には並列スケルトンライブラリ SkeTo [15] の非公開最新版を用いている．実験環境はギガビットイーサネットでつながれた PC クラスタである．各計算機の構成は CPU が dual Xeon 2.8 GHz，メモリ 2 GB，コンパイラが GCC 4.1.2 および MPICH 1.2.7-p1，OS が Linux 2.6.18-AMD64 である．各計算機はデュアルコアであるが，実験に際しては各々1 コアずつしか用いていない．また，比較のために，各問題に対する逐次プログラムを用意し，その実行時間も計測した．なお，ここでの実行時間にはデータの初期化や分散に要する時間は含まない．

表 3　並列化器の生成したプログラムの実行時間（ミリ秒）
Table 3　Execution times of generated codes. (unit: millisecond)

	逐次	1 台	2 台	4 台	8 台	16 台	32 台	64 台
max	313	697	369	186	94	52	32	118
mps	329	540	270	137	68	34	26	107
mss	353	1,299	652	331	165	98	50	39
atoi	414	1,599	803	401	200	106	59	38

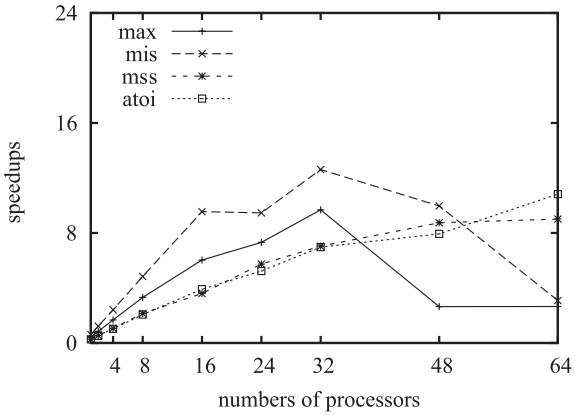

図 9　逐次プログラムに対する速度向上
Fig. 9　Speedups with respect to the numbers of processors against sequential implementations.

表 3 に生成された各プログラムの実行時間を記す．各々の並列プログラムについて 20 回ずつ実行時間を計測し，その最低値をそのプログラムの実行時間としている．また，生成された各並列プログラムの逐次プログラムに対する速度向上を**図 9** に示す．

まず，いずれのプログラムも 32 台程度まではプロセッサ数に対してほぼ線形の台数効果を示していることが分かる．しかし，比較的実行に時間のかかるプログラム（*mss* と *atoi*）では 64 台まで台数効果を示している一方，実行に時間のかからないプログラム（*max* と *mis*）では 48 台や 64 台ではむしろ遅くなっている．これは通信に多くの時間を要してしまっているためだと考えている．

1 台での実行時間は，それぞれ逐次プログラムに対して 2 倍から 4 倍程度遅い．この理由

には，SkeTo のオーバヘッドがあげられるが，並列化そのものによるオーバヘッドもある．たとえば最大接頭部和問題（*mps*）では，逐次プログラムは最大接頭部和の値のみを計算，保持すればよいのに対し，並列プログラムでは最大接頭部和の値および配列の要素和を計算，保持している．このように，一般に並列プログラムは逐次プログラムよりも多くの値を計算することになる．

以上の議論からも分かるとおり，今回我々が生成した並列プログラムは十分に効率の良いものだとはいい難い．並列化結果から実際にいかに効率の良い並列プログラムを出力するかは今後の課題である．

6. 議　　論

本論文では，特に配列を走査して値を求める計算（リダクション）を行うループに対する新しい自動並列化手法を提案した．提案手法の要旨は，並列プログラム候補を列挙しその中から適切なものを発見するということである．並列化の鍵は第三準同型定理によって並列化可能な関数集合を特定する点である．実際，いくつかの例について，並列化可能な関数集合からは少数の候補を考えるだけで並列プログラムを得ることができた．提案手法の有用性を確認するため，自動並列化のための領域特化言語を設計し，限定記号除去に基づいた自動並列化器を実装した．我々の並列化器は以前の手法[16),22)]では並列化できなかったいくつかの例を並列化することができた．

6.1 関 連 研 究

ループの自動並列化については多くの研究がある．しかし，その多くは，完全に独立した計算などの既知の並列処理可能な計算をいかにプログラム中から発見するか，またはいかにそのようなコードを多重ループの平坦化などによって明示的なものとするか，という点に重点が置かれてきた[12),18)–20)]．リダクションについては，その重要性は広く認識されているにもかかわらず[7)]，和や積など特定の場合のみについての議論が中心であり，並列に計算できる演算を自動的に導出する研究は少なかった．しかし，並列プログラムの効率はプログラム中のどれだけの部分が並列化されているかに大きく依存するため，複雑なリダクションに対する自動並列化を議論することも重要である．

リダクションに対し，並列に計算できる演算を自動的に導出する研究については，我々の知る限り大きく分けて 2 種類ある．1 つは第三準同型定理に基づくもので，もう 1 つは関数合成に基づくものである．

本論文では第三準同型定理に基づいて自動並列化手法を構築した．第三準同型定理の長所は，右方向と左方向 2 つのプログラムを用意することで，自動並列化のための情報を並列化器に与えることができる点にある．そのため，1 つの逐次プログラムからに比べて容易に自動並列化を行えると期待できる．我々は以前にも第三準同型定理に基づいた自動並列化手法を提案した[16),22)]．しかし，この手法には 2 つの大きな問題点があった．1 つは計算の結果が配列の長さに大きく依存するような計算を並列化できないこと，もう 1 つは並列化の結果として得られるプログラムの効率が一般には良くないことである．本手法はこの 2 点の問題点を解決することを意図して考案された．Geser ら[10)]も第三準同型定理に基づいた自動並列化手法を提案している．彼らの手法は，右方向の定義と左方向の定義の両方を表現したプログラムを生成し（彼らはこの処理を「anti-unification」と呼んでいる），それが確かにリスト準同型であることを確認するものである．我々は，彼らの手法を方法論として一般化し，候補生成とその性質の確認による自動並列化の枠組みを提案した．

ループの 1 回の計算を関数と見なしたとき，その関数合成が効率的に計算できれば，ループ全体を並列に計算することができる．この着想に基づいて，いくつかの自動並列化手法が提案されている．Callahan[3)]は特に差分方程式のようなプログラムについて並列化手法を示した．また，Xu ら[21)]は結合則や分配則などの代数的な性質に基づいた自動並列化手法を提案した．より一般的なプログラムに対する取り組みとしては Fisher ら[9)]のものがある．彼らは条件式を含むプログラムの自動並列化手法を提案した．これらの手法では，1 つの逐次プログラムから並列プログラムを得ようと試みるため，問題の領域を大きく制限する，ユーザが何らかの性質を保証する，複雑な実装を行う，などが必要であった．我々の研究はこれらの手法に強く動機づけられており，2 つの逐次プログラムを与えることでより簡単な自動並列化手法を構築できないか，という着想に基づいている．

6.2 今後の課題

ここでは，我々の実装では自動並列化に失敗する例をいくつかあげることで，今後の課題の議論に代える．また，それぞれの例について，Morita らの手法[16),22)]で扱えるか否かについても述べる．結論を先に述べると，我々の手法で扱えない多くの例は Morita らの手法で扱うことができる．そのため，両手法を組み合わせることでより強力な自動並列化手法を構築できると期待している．

用いる限定記号除去手法による問題

最大接頭部積問題[22)]を解く関数 *mpp* は以下の左方向のプログラムによって定義することができる．

$$mpp([a]) \quad = a$$

$mpp([a] +\!\!+ x) = a \uparrow (a \times mpp(x)) \uparrow (a \times npp(x))$
$npp([a]) \quad = a$
$npp([a] +\!\!+ x) = a \downarrow (a \times npp(x)) \downarrow (a \times mpp(x))$
$product([a]) \quad = a$
$product([a] +\!\!+ x) = a \times product(x)$

負数どうしの乗算によって正数が生じる場合があるため，最小接頭部積を計算する関数 *npp* が必要となっている．また，関数 *product* は両方向の定義を与えるために必要である．

関数 *mpp* は現在の実装では並列化することができない．なぜなら，限定記号除去による結合性の証明の過程で変数の自乗の項が現れ，実装した限定記号除去手法の適用範囲から外れてしまうためである．この問題は，たとえば cylindrical algebraic decomposition [5] などの，より強力な限定記号除去アルゴリズムを用いることで解決できる．より正確にいえば，図 5 に示した言語で記述されたプログラムに対しては，4.2.3 項の手続きに従って得られる式の真偽は cylindrical algebraic decomposition によって必ず決定できる．しかし，強力な限定記号除去アルゴリズムは一般により長い処理時間を要し，かつ，上のプログラムからは 730 もの候補が生成されるため，現実的な時間では処理を終えられないことも危惧される．

用いる限定記号除去手法によって並列化可能な問題の範囲や並列化の効率が変化するのは我々の手法も Morita らの手法も同様である．実際，Morita らの手法でも，最大接頭部積問題は Fourier-Motzkin elimination を用いた実装[16] では扱うことができず，より強力な限定記号除去手法を用いることで並列化可能となった[22] ことが報告されている．

定義域に制限がある再帰関数に関する問題

以下で定義される関数 *odd* は配列長が奇数ならば真，偶数ならば偽を返す．ただし，我々の言語では整数値しか用いることができないため，真を 1 で，偽を 0 で表現している．

$odd([a]) \quad = 1$
$odd([a] +\!\!+ x) = \mathsf{if}\ odd(x) = 1\ \mathsf{then}\ 0\ \mathsf{else}\ 1$

関数 *odd* の並列化の際に証明しなければならないのは，端的には以下の演算子 $\odot$ の結合性である．

$a \odot b = \mathsf{if}\ a = b\ \mathsf{then}\ 0\ \mathsf{else}\ 1$

我々の実装では実数上の限定記号除去によって適合性検査を行う．しかし，この演算子 $\odot$ は実数上や整数上では結合則を満たさない．たとえば，$(1 \odot 2) \odot 0 = 1 \odot 0 = 1$ であるのに対し，$1 \odot (2 \odot 0) = 1 \odot 1 = 0$ である．よって，我々の実装では関数 *odd* の並列化は失敗する．しかし，引数が 0 または 1 に限られる場合，$\odot$ は結合的である．そのため，実際には $\odot$ を *odd* の並列計算に用いることには問題がない．以上のように，より正確な適合性検査には演算子の定義域を知ることが必要である．特に真偽値や整数値をとる演算については，それぞれの定義域に応じた定理証明手法を用いるのが効果的である．

Morita らの手法では，このような問題は生じにくい．なぜなら，Morita らの手法では並列化対象の再帰関数の値域上で定義される演算子を導出するためである．また，関数 *odd* はその値が配列長に大きく依存するため Morita らの手法では扱うことができない．

構文上の差異による問題

以下で定義される関数 *psum* は，配列中の正の数の和を求める．

$psum([a]) \quad = \mathsf{if}\ a > 0\ \mathsf{then}\ a\ \mathsf{else}\ 0$
$psum([a] +\!\!+ x) = \mathsf{if}\ a > 0\ \mathsf{then}\ a + psum(x)\ \mathsf{else}\ psum(x)$

我々の実装では，この関数定義からの自動並列化は，適切な候補が生成さないために失敗する．一方，以下の定義からであれば自動並列化を行うことができる．

$psum([a]) \quad = \mathsf{if}\ a > 0\ \mathsf{then}\ a\ \mathsf{else}\ 0$
$psum([a] +\!\!+ x) = (\mathsf{if}\ a > 0\ \mathsf{then}\ a\ \mathsf{else}\ 0) + psum(x)$

このように，我々の手法による並列化の成否はユーザによる関数記述に大きく依存する．効果的な並列化のためには，結合則や分配則などの性質を用いることで，できる限り多くの候補を生成する必要がある．

また，似た例として先にあげた関数 *odd* を再考する．実は，我々の実装でも，以下の式による *odd* の定義からであれば自動並列化を行うことができる．

$odd([a]) \quad = 1$
$odd([a] +\!\!+ x) = \mathsf{if}\ (odd(x) = 1 \wedge odd(x) = 1)\ \mathsf{then}\ 0\ \mathsf{else}\ 1$

このように，一見冗長な定義を用いることによって，生成する候補が増え並列化が成功することも少なくない．しかし，単純に生成候補数を増やすのは並列化に要する時間などの点で現実的でない．

Morita らの手法では並列化の成否はプログラムの表現にほとんど依存しない．この点は Morita らの手法の大きな特徴の 1 つである．

値に関連のある再帰関数に関する問題

以下で定義される関数 *sum1* および *sum2* は配列の要素和を求める．

$sum1([a]) \quad = a$
$sum1([a] +\!\!+ x) = a + sum2(x)$
$sum2([a]) \quad = a$

$sum2([a] +\!\!+ x) = a + sum1(x)$

配列の要素和を求める並列プログラムを得るのは容易である．しかし，関数群 $\{sum1, sum2\}$ は我々の手法では並列化に失敗する．

並列化の際に証明が必要となるのは，たとえば以下の式である．

$a + sum1(x) + sum2(y) = a + sum2(x) + sum2(y)$

我々の手法では，関数呼び出しは任意の値をとりうると見なしているため，上式を証明できない．しかし，実際には $sum1(x)$ の値はつねに $sum2(x)$ の値に一致するため，上式は成り立つ．このように，複数の再帰関数の値に強い依存関係がある場合，我々の手法では証明を行うことが難しい．

先に述べたとおり，Morita らの手法では関数の値域上でのみ定義される演算子を導出するためこのような問題は起こりにくい．

謝辞　第 29 回 TRS meeting の参加者の皆様との意義ある議論に対し感謝する．本論文の第 1 著者は日本学術振興会特別研究員として科学研究費補助金（課題番号 20・2411）の補助を受けている．

参 考 文 献

1) Bentley, J.: *Programming Pearls*, ACM, New York, NY, USA (1986).
2) Bird, R.S.: An Introduction to the Theory of Lists, *Logic of Programming and Calculi of Discrete Design*, NATO ASI Series F, Vol.36, pp.3–42, Springer (1987).
3) Callahan, D.: Recognizing and Parallelizing Bounded Recurrences, *Languages and Compilers for Parallel Computing, 4th International Workshop, Santa Clara, California, USA, August 7-9, 1991, Proceedings*, Lecture Notes in Computer Science, Vol.589, pp.169–185, Springer (1992).
4) Chin, W.-N.: Towards an automated tupling strategy, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93*, Copenhagen, Denmark, June 14-16, pp.119–132, ACM (1993).
5) Collins, G.E.: Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition, *Automata Theory and Formal Languages, 2nd GI Conference*, Kaiserslautern, May 20-23, 1975, Lecture Notes in Computer Science, Vol.33, pp.134–183, Springer (1975).
6) Dantzig, G.B. and Eaves, B.C.: Fourier-Motzkin Elimination and Its Dual, *Journal of Combinatorial Theory, Series A*, Vol.14, No.3, pp.288–297 (1973).
7) Deitz, S.J., Callahan, D., Chamberlain, B.L. and Snyder, L.: Global-view abstractions for user-defined reductions and scans, *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006*, New York, New York, USA, March 29-31, 2006, pp.40–47, ACM (2006).
8) Dolzmann, A. and Sturm, T.: Simplification of Quantifier-Free Formulae over Ordered Fields, *Journal of Symboric Computation*, Vol.24, No.2, pp.209–231 (1997).
9) Fisher, A.L. and Ghuloum, A.M.: Parallelizing Complex Scans and Reductions, *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* (*PLDI*), Orlando, Florida, June 20-24, 1994, pp.135–146, ACM (1994).
10) Geser, A. and Gorlatch, S.: Parallelizing functional programs by generalization, *Journal of Functional Programming*, Vol.9, No.6, pp.649–673 (1999).
11) Gibbons, J.: The Third Homomorphism Theorem, *Journal of Functional Programming*, Vol.6, No.4, pp.657–665 (1996).
12) Größlinger, A., Griebl, M. and Lengauer, C.: Quantifier elimination in automatic loop parallelization, *Journal of Symbolic Computation*, Vol.41, No.11, pp.1206–1221 (2006).
13) Hu, Z., Iwasaki, H., Takeichi, M. and Takano, A.: Tupling Calculation Eliminates Multiple Data Traversals, *Proc. 2nd ACM SIGPLAN International Conference on Functional Programming, ICFP'97*, Amsterdam, The Netherlands, pp.164–175, ACM (1997).
14) Loos, R. and Weispfenning, V.: Applying Linear Quantifier Elimination, *The Computer Journal*, Vol.36, No.5, pp.450–462 (1993).
15) Matsuzaki, K., Iwasaki, H., Emoto, K. and Hu, Z.: A library of constructive skeletons for sequential style of parallel programming, *Proc. 1st International Conference on Scalable Information Systems, Infoscale 2006*, Hong Kong, May 30-June 1, 2006, ACM International Conference Proceeding Series, Vol.152, p.13, ACM (2006).
16) Morita, K., Morihata, A., Matsuzaki, K., Hu, Z. and Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs, *Proc. ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, California, USA, June 10-13, 2007, pp.146–155, ACM (2007).
17) Peyton Jones, S. (Ed.): *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press (2003).
18) Pottenger, W.M.: The Role of Associativity and Commutativity in the Detection and Transformation of Loop-level Parallelism, *International Conference on Supercomputing*, ACM, pp.188–195 (1998).
19) Redon, X. and Feautrier, P.: Detection of Recurrences in Sequential Programs with Loops, *PARLE '93, Parallel Architectures and Languages Europe, 5th International PARLE Conference, Proceedings*, Munich, Germany, June 14-17, 1993, Lecture Notes in Computer Science, Vol.694, pp.132–145, Springer (1993).
20) Suganuma, T., Komatsu, H. and Nakatani, T.: Detection and Global Optimization

of Reduction Operations for Distributed Parallel Machines, *International Conference on Supercomputing*, pp.18–25, ACM (1996).
21) Xu, D.N., Khoo, S.-C. and Hu, Z.: PType System: A Featherweight Parallelizability Detector, *Programming Languages and Systems: 2nd Asian Symposium, APLAS 2004, Proceedings*, Taipei, Taiwan, November 4-6, 2004, Lecture Notes in Computer Science, Vol.3302, pp.197–212, Springer (2004).
22) 森田和孝：右逆関数の自動導出によるプログラムの並列化に関する研究，修士論文，東京大学大学院情報理工学系研究科 (2007).

(平成 20 年 9 月 28 日受付)
(平成 20 年 12 月 20 日採録)

森畑　明昌

1981 年生．2004 年東京大学工学部計数工学科卒業．2006 年同大学大学院情報理工学系研究科修士課程修了．プログラム変換，アルゴリズム導出等に興味を持つ．日本ソフトウェア科学会会員．

松崎　公紀（正会員）

1979 年生．2001 年東京大学工学部計数工学科卒業．2003 年同大学大学院情報理工学系研究科修士課程修了．2005 年同研究科博士課程中退．同年より同研究科助手，2007 年より助教となり現在に至る．博士（情報理工学）．並列プログラミング，アルゴリズム導出等に興味を持つ．日本ソフトウェア科学会会員．

胡　　振江（正会員）

1966 年生．1988 年中国上海交通大学計算機科学系を卒業．1996 年東京大学大学院工学系研究科情報工学専攻博士課程修了．同年日本学術振興会特別研究員を経て，1997 年東京大学大学院工学系研究科情報工学専攻助手，同年 10 月同専攻講師，2000 年同専攻准教授，2008 年より国立情報学研究所教授．博士（工学）．プログラミング言語，関数プログラミング，ソフトウェア工学，並列プログラミング等に興味を持つ．

武市　正人（正会員）

1948 年生．1972 年東京大学工学部助手，講師，電気通信大学講師，助教授，東京大学工学部助教授を経て 1993 年東京大学大学院工学系研究科教授（情報処理工学講座），2001 年より同大学大学院情報理工学系研究科教授，現在に至る．2003 年より日本学術会議会員，現在に至る．工学博士．プログラミング言語，関数プログラミング，構造化文書処理の研究・教育に従事．日本ソフトウェア科学会，日本応用数理学会，ACM 各会員．