情報処理学会論文誌　プログラミング　Vol.6　No.4　38–49 (Dec. 2013)

木上のスケルトン並列プログラミングのための演算子生成器

佐藤 重幸[1,a)]　松崎 公紀[2]

受付日 2013年4月12日, 採録日 2013年9月20日

概要：木上の並列計算を抽象化および構造化する手段として，木スケルトンというものが存在する．著者らは，木スケルトンを C++上の高階関数ライブラリとして実装してきている．この木スケルトンは，効率的な並列計算を保証するために，逐次計算において使われる演算子に加えて，補演算子も要求する．この補演算子の導出は一般に難しく，木スケルトンの利用への大きな障害となっている．そこで本研究では，既存の補演算子導出法に基づいた演算子生成器を，C コンパイラ拡張として実装した．本システムは，木スケルトンを隠蔽することで，利用者が木スケルトンの複雑さに触れることを防ぐ．利用者は逐次的な再帰関数を C で記述するだけで，暗黙かつ単純に木スケルトンを利用できるようになる．したがって，本システムは，木上のスケルトン並列プログラミングの難しさを首尾よく解消する．

キーワード：木スケルトン，演算子，双線型計算，並列プログラミング，コンパイラ

An Operator Generator for Skeletal Programming on Trees

Shigeyuki Sato[1,a)]　Kiminori Matsuzaki[2]

Received: April 12, 2013, Accepted: September 20, 2013

Abstract: Tree skeletons are known as a way of abstracting and structuring parallel computation on trees. We have been implementing tree skeletons as a library of higher-order functions in C++. For efficient parallel computing, our tree skeletons necessitate auxiliary operators as well as operators used in sequential computation. Since an auxiliary operator is difficult to derive in general, our tree skeletons are too difficult for non-expert programmers to use. To overcome this difficulty, on the theoretical basis of existing work, we have developed an operator generator in the form of an extension to a C compiler. By hiding a tree skeleton itself, our implementation prevents users from touching its complicatedness; then they have only to describe recursive functions in C to use a tree skeleton simply and implicitly. Our implementation resolves the difficulty of skeletal programming on trees successfully.

Keywords: tree skeleton, operator, bilinear computation, parallel programming, compilers

1. 動機と貢献

構造化プログラミング [5] とは，goto のような無制限の制御構造を使う代わりに，if や while のように制限された制御構造を組み合わせることでプログラミングを行うことであり，良いプログラミング手法であると広く認められて久しい．これと同様の考え方を，並列プログラムに適用したのが，スケルトン並列プログラミング [4] である．このとき組み合わせる対象は，単なる制御構造ではなく，Algorithmic Skeleton（もしくは単にスケルトン）と呼ばれる，並列計算を抽象化したパターンである．一般に，スケルトンはそれが操作するデータ構造に合わせて設計される．

まずは，単純なリスト[*1]上のスケルトン [16] について考える．リストを逐次的に畳み込む計算 foldr は，

$$\mathrm{foldr}(\oslash, e, [x_1, \ldots, x_n]) = x_1 \oslash (x_2 \oslash (\cdots (x_n \oslash e) \cdots))$$

と定義できる．この直観的な意味は，右結合演算子 $\oslash$ を用

[1] 電気通信大学大学院情報理工学研究科
Graduate School of Informatics and Engineering, The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[2] 高知工科大学情報学群
School of Information, Kochi University of Technology, Kami, Kochi 782–8502, Japan
[a)] sato@ipl.cs.uec.ac.jp

[*1] スケルトンの文脈におけるリストとは，実質 1 次元配列である．

© 2013 Information Processing Society of Japan

いて各要素をつなぎ，リスト終端（空リスト）を e で置き換えた後に，その式を評価することである．これの並列版である reduce スケルトンは，引数として受け取る演算子が右結合演算子 $\oslash$ から結合的演算子 $\oplus$ に変わるだけで，直観的な意味は foldr とほぼ同じである．しかし，結合性から括弧付けを自由に変更することが可能になるため，分割統治による効率的な並列計算が実現できる．したがって，リストスケルトンの reduce を利用する場合は，演算子の結合性の保証にさえ気をつければ，通常の逐次プログラムと同様にプログラミングできる．

次に，本研究の対象とする木上のスケルトン [6], [8], [17] について考える．ここでは，簡単のため全 2 分木とする．ノード n が持つ要素値を $n.v$，左右の子をそれぞれ $n.l$ と $n.r$ と定義したとき，木を畳み込む計算 fold は次のように定義できる．

$$\mathrm{fold}(k_E, k_B, n) = \mathrm{fold}'(n) \quad \text{s.t.}$$

$$\mathrm{fold}'(n) = \begin{cases} k_E(n.v) & (n \text{ が葉}) \\ k_B(\mathrm{fold}'(n.l), n.v, \mathrm{fold}'(n.r)) & (\text{さもなくば}) \end{cases}$$

この fold の直観的な意味は，リストと同様に，葉ノードと中間ノードをそれぞれ k_E と k_B で置き換えるように要素値をつなぎ，その式木をボトムアップに評価することである．以降，計算を特徴付ける k_E と k_B を，主演算子と呼ぶ．これの並列版である木スケルトンの reduce は，主演算子に加えて，効率的な並列計算を保証するために，補演算子を要求する．この補演算子は，直観的には，木の内側について計算を進めて，最終的に主演算子と同等の結果をもたらすものである．

この補演算子を必要とすることは，リストスケルトンに比べて，木スケルトンをとても使いにくくする．補演算子は，その定義から主演算子と独立に用意することはできない．満たすべき代数的性質も，リストスケルトンのように演算子単独の結合性ではなく，より複雑なものである．そして何より，主演算子から一般に補演算子を導出する手法は存在していない．したがって，木スケルトンの利用者は，主演算子を定義した後に，複雑な性質を満たす補演算子をすべて正しく手作業で導出しなければならない．これは，木上の並列プログラムの導出法を熟知した専門家でなければ不可能な作業であり，本来の目標である「良いプログラミング手法」からは程遠いものである．

この補演算子導出の労力を減らすために，Matsuzaki ら [10] は，主演算子から補演算子を系統的に導出する方法を与えた．そして，その有用性と実現可能性を示すために，演算子生成器の試験的実装も与えた．これによって，木スケルトンの利用者が，限定された範囲について，補演算子の導出を手作業で行う必要はなくなった．しかし，木上のスケルトン並列プログラミングシステムの実装という観点からは，彼らの実装には設計上の問題が 2 つある．

第 1 の問題は，彼らの演算子生成器の入力言語として，C++に似た専用の独自言語を採用している点である．そのため，入力である木上の逐次的な再帰関数自体に対して，一般的なコンパイル時検査をかけることや，実行してデバッグすることができない．また，利用者の立場からすれば，単に木上の再帰関数を書くためだけに，新しい言語の構文を覚えるのは手間である．入力言語としては，一般によく知られ利用されている言語を採用するべきである．

第 2 の，本研究で最も重大視している問題は，利用者に木スケルトンを直接利用させている点である．利用者の視点では，生成された意味不明のいくつもの演算子を，やはり意味不明の API を持つ木スケルトンに，正しく与えなければならない．演算子の仕様記述が抽象化され，演算子導出が自動化されたのであれば，もはや木スケルトン API は，利用者が触るべきでも知るべきでもない実装の詳細でしかない．よって，演算子生成器には，木スケルトン API を隠蔽する抽象化も必要である．

そこで本研究では，Matsuzaki らの手法を基礎として，木スケルトンの利用自体を完全に隠蔽するように設計された演算子生成器を，C コンパイラ拡張の形で実装した．これによって，Matsuzaki らの手法が適用可能な範囲については，2 分木上の単純な再帰関数を C で記述するだけで，木スケルトンの実装が透過的に利用可能になる．したがって，利用者は，木スケルトンの持つ複雑さに悩まされずに，木上のスケルトン並列プログラミングを行える．

本論文の貢献は，次のようにまとめられる．

- 木上のスケルトン並列プログラミングを容易にする演算子生成器を，既存研究 [10] に基づいて，C コンパイラ拡張として COINS [*2] 上に実装した（4.1 節）．
- 我々の実装は，これまでは考慮されてこなかった木スケルトン自体の抽象化を行うことで，利用者に木スケルトンの複雑さを曝すことがないように設計されている（3 章）．この設計は，多分木スケルトンに拡張しても，複雑さを導入しない（6 章）．
- 演算子生成器とともに，木スケルトンライブラリも実装し（4.2 節），簡単な動作実験を行った（5 章）．

2. 木スケルトンの補演算子とその導出法

本研究では，Matsuzaki ら [10] が提案した補演算子導出法を用いる．本章では，まず，スケルトン並列プログラミングという概念について改めて説明する．次に，Matsuzaki [8] の木スケルトンにおける補演算子の役割と性質を簡単に説明する．最後に，彼らの補演算子導出法を非形式的に紹介する．これらの詳細については，文献 [8], [10] を参照されたい．

[*2] http://coins-compiler.sourceforge.jp/

© 2013 Information Processing Society of Japan

2.1　スケルトン並列プログラミング

1 章でも述べたが，スケルトンとは，並列計算のパターンである．注意すべきことは，我々の対象とするスケルトンが，抽象的なパターンであることである．スケルトンは，どのような結果を返すか，という数学的な意味は定義されているが，どのように計算するか，という操作的な意味は定義しない．よって，reduce は foldr や fold と同じ結果を返すと説明ができるが，具体的な計算は対象とするデータによって決まる．

スケルトンは，漸近的な台数効果について一定の保証を与える．たとえば，1 章で紹介したリストスケルトン reduce は，$\oplus$ の結合性から，分割統治による並列計算が可能になり，線型の台数効果を保証できる．ここで重要な役割を果たすのは，演算子の代数的性質である．$\oplus$ の結合性が，括弧の付け替えを可能にし，foldr の効率的な並列化を実現する．言い換えると，スケルトンは，並列化という概念を，演算子の代数的性質で表現している．

スケルトンにおいて，もう 1 つ重要なのは，データ構造を抽象化することである．リストスケルトンは，foldr の走査対象である Cons リストを走査して並列計算を行うのではない．リストスケルトン用の並列データ構造を入力として，その上を走査する．並列データ構造は，基本的にスケルトンによる並列操作しか許可しない．スケルトン並列プログラミングとは，スケルトンという抽象並列操作と，抽象並列データ構造の組合せによって，並列プログラムを構築することである．

2.2　木スケルトンの補演算子

リストスケルトンでは，括弧を付け替えるという単純な代数的操作によって，台数効果が得られるが，木スケルトンでは，より複雑な操作が要る．補演算子は，その並列化のための操作を定式化する演算子である．これを補演算子と呼ぶ理由は，それらが計算結果を与える fold において不必要だからである．ここで重要なのは，補演算子は，代数的性質のみのよって形式的に定義されるということである（詳細は付録 A.1 参照）．補演算子は，主演算子との間で定義された代数的性質によって，主演算子に対応した意味付けがなされる．そして，補演算子の間に定義された代数的性質によって，代数的操作による並列化が実現される．

木スケルトンの補演算子は，直観的には，ボトムアップ演算と等価な計算を木の内側から進めるための演算子である．それを導入する利点は，ひとえに，より自由度が高い分割統治が可能になることである．補演算子によって，クリティカルパスが分割され，木を分散させたとしてもわずかな通信で負荷分散できる．リストのような極端な形の入力木でも，共有メモリ環境でも分散メモリ環境でも，漸近計算量における線型の台数効果が保証される．

補演算子の中でも特に本質的な働きを持つのは，木の「中間」を潰す役割を持つ縮約演算子である．具体的には，左の子が葉ノードで右の子が中間ノードであるような親子のノードを，1 つの中間ノードに潰す演算子 ψ_L が縮約演算子である．木のデータ構成子 *Branch* と *Leaf* を用いて，記号的に ψ_L の計算を表すと次のようになる．

$$Branch(Leaf(v_L), v, Branch(t_L, v_R, t_R)) \xrightarrow{\psi_L} Branch(t_L, v', t_R)$$

ψ_R は，ψ_L と左右対称の演算子である．これらの縮約演算子は，潰す対象の「中間」をどこから選んで計算を進めても，最終的な結果が変わらないことが保証されなければならない．この性質は，代数的には，一種の結合性である．実際，リストを細長い木と見なせば，それをどこから潰してもよいというのは，結合的演算子でリストを畳み込むときに括弧付けを自由に変更できることと等価である．

結合的演算というものは同一定義域上の演算なので，一般に左右オペランド型と返値型が一致していなければならない．縮約演算子も同等の型制約がある．これは単なる型の問題でない．どこからでも計算可能にするためには，逐次計算では捨てることができる計算の途中状態を残したうえで，つねに整合をとらなければならないという並列計算の普遍的な性質に由来する．よって，元の入力とともに，計算の途中状態を保持するようなデータ型へと変換する必要がある．その役割を担うのが，木の要素の型を縮約演算の定義域の型へと変換するリフト演算子 τ である．さらに，縮約演算子によって計算が進んだ後に，リフト先の型から fold と同等の結果を得なければならない．よって，リフトされた木上でのボトムアップ演算子 ϕ が必要になる．

縮約演算子，リフト演算子，リフトされたボトムアップ演算子，この 3 種類によって補演算子が構成される．それぞれの演算子の形式的定義は，付録 A.1 に示す．

2.3　補演算子の系統的導出

2.3.1　双線型計算

Matsuzaki ら [10] の補演算子導出法は，ある種の計算クラスを対象とする．それは，fold のボトムアップ演算が，左右の部分木の計算結果の線型式である計算である．形式的には，まず，ある中間ノード n と，その左右の部分木 t_L と t_R を考える．t_L と t_R についての結果をそれぞれ $\mathbf{x}_L$ と $\mathbf{x}_R$ として，n を根とする部分木の結果を $\mathbf{x}$ としたときに，次の関係が成り立つことである．

$$\mathbf{x} = A_R\mathbf{x}_L = A_L\mathbf{x}_R \tag{1}$$

ここで，n が持つ要素を v としたときに，A_L は $\mathbf{x}_L$ の成分と v からなる行列であり，A_R は $\mathbf{x}_R$ の成分と v からなる行列である．式 (1) を展開すると，次のようになる．

© 2013 Information Processing Society of Japan

$$\begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix} = \begin{pmatrix} a^R_{11}l_1 + \cdots + a^R_{1k}l_k \\ \vdots \\ a^R_{k1}l_1 + \cdots + a^R_{kk}l_k \end{pmatrix} = \begin{pmatrix} a^L_{11}r_1 + \cdots + a^L_{1k}r_k \\ \vdots \\ a^L_{k1}r_1 + \cdots + a^L_{kk}r_k \end{pmatrix}$$

ここで，$\mathbf{x} = (x_1 \ \cdots \ x_k)^T$，$\mathbf{x}_L = (l_1 \ \cdots \ l_k)^T$，$\mathbf{x}_R = (r_1 \ \cdots \ r_k)^T$ である．式 (1) の形から，A_L の成分 a^L_{ij} は，$l_1, \ldots, l_k$ についての多変数 1 次式であり，A_R の成分 a^R_{ij} は，$r_1, \ldots, r_k$ についての多変数 1 次式である．

単に式 (1) だけでは，定数項を持つ多変数 1 次式，たとえば $x = alr + b$ を表現できないように見える．しかし，拡大係数行列を用いて，定数項を係数行列に含めれば，定数項を持つ多変数 1 次式を式 (1) で表現できる．

我々は，この計算クラスを 2 分木上の再帰的双線型計算（もしくは単に双線型計算）と呼ぶことにする．

2.3.2　双線型計算における補演算子

式 (1) を木計算として考えると，リストのような一直線の木に対する計算が，葉の結果を $\mathbf{x}_l$ としたときに，行列乗算の連鎖積 $A_1A_2 \cdots A_h\mathbf{x}_l$ と表現できることを意味する．木上のボトムアップ演算は，この連鎖積を右から順に行列ベクトル乗算を繰り返して計算することに相当する．行列乗算の結合性から，この連鎖積における行列乗算はどこから計算してもよい．よって，木の中間を計算する縮約演算は，行列行列乗算となる．

縮約演算が行列乗算なので，その計算の途中状態である行列と，元の入力の情報を両方が計算に必要になる．よって，リフト演算子 τ は，要素型 a から，a と成分型が a である行列 $\mathcal{M}_a$ の直積 $(a, \mathcal{M}_a)$ へリフトしなければならない．行列連鎖積に影響を与えない行列は，単位行列 I なので，τ は，ノードの要素 v から v と I の組 (v, I) を構築する演算となる．リフトされた木上のボトムアップ演算子は，主演算子と行列ベクトル乗算の合成となる．たとえば，ϕ_B は，k_B の演算結果のベクトルと，リフトされたノードが持つ行列との積をとる演算となる．この行列ベクトル乗算は，縮約演算によって行列の形で蓄積されていた木の中間の計算を，まとめて適用する役割を持つ．その行列ベクトル積は，簡約された木の中間における計算の作用が，ボトムアップ計算の作用に合流した結果である．

2.3.3　可換半環上で一般化された補演算子導出

以上の説明のとおり，双線型計算では，各補演算子が行列演算を用いた特定の形式になる．式 (1) の行列を用いて，形式的に 3 種の補演算子の代表を行列形式で定義すると，次のようになる．

$$\psi_L(\mathbf{x}_L, (v, M), (v', M')) = (v', MA_LM')$$
$$\phi_B(\mathbf{x}_L, (v, M), \mathbf{x}_R) = MA_L\mathbf{x}_R$$
$$\tau(v) = (v, I)$$

ここで，左辺に出現する v や $\mathbf{x}_L$ は，右辺では A_L やの中に埋め込まれている点に注意されたい．

ここで重要なのは，ここでの行列演算は可換半環上で一般化できることである[*3]．一般に，$(S, +, \times, 0, 1)$ が可換半環であるとは，加法 $+$ と乗法 $\times$ に結合性 $(a*(b*c) = (a*b)*c)$ と，可換性 $(a * b = b * a)$ が成り立つこと．そして，$\times$ が $+$ に対して分配的 $(a \times (b + c) = a \times b + a \times c$ かつ $(a+b)\times c = a\times c+b\times c)$ であること．0 と 1 が，それぞれ $+$ と $\times$ の単位元 $(0+a = a+0 = a$ かつ $1\times a = a\times 1 = a)$ となること．0 が $\times$ の吸収元 $(0 \times a = a \times 0 = 0)$ となること．これらすべての代数的性質が，S 上の任意の元について成り立つことである．可換半環の例としては，通常の環 $(\mathbb{Q}, +, \cdot, 0, 1)$，熱帯半環 $(\mathbb{Q}, \max, +, -\infty, 0)$，ブール半環 $(\{0, 1\}, \vee, \wedge, 0, 1)$ があげられる．したがって，それらの代数系における双線型計算であれば，統一的に行列演算によって補演算子を定義できる．

3.　システムの設計と利用例

我々は，1 章で述べた既存実装の問題点を解決するシステムとして，演算子生成器と，生成された演算子で計算する木スケルトンを実装した．本章では，それらの設計と具体的な利用例を説明する．

3.1　基本設計

1 章で指摘した 2 つの問題点を解決するために，それぞれの問題点に対して次の設計方針をとった．

- 標準の C を入力言語とし，ディレクティブを利用した C コンパイラ拡張として演算子生成器を実装する．
- 木スケルトンを直接利用する部分を隠蔽する入口関数を，演算子とともに自動生成する．

第 1 の方針により，言語の学習コストが低くなり，単独でのコンパイル時検査や実行テストが可能な逐次的な再帰関数を計算の仕様として受け取ることが可能になる．付加的情報は，プリプロセッサディレクティブの形で記述させることで，C の標準構文を遵守した．ただし，演算子導出法自体の制限から，並列化対象の再帰関数には表現上の制約がある．我々の目的は，C プログラムの並列化ではない．よって，本システムでは，対象関数について C の意味を厳格には遵守せず，抽象アルゴリズムのレベルで逐次版と並列版が同等な結果を返すことしか保証しない．

第 2 の方針により，本システムの利用者は，木スケルトンを直接利用することがなくなる．利用者の視点では，並列化対象の関数の代わりに，並列木データ構造に入口関数を適用するだけで並列計算が達成される．木スケルトンの API を利用者から隠蔽するこの設計は，多分木の場合でも利用者に負担を強いることがない．この点については，6 章で詳しく議論する．

[*3] 厳密には，式 (1) の形式が，暗黙に可換半環を要求する．

© 2013 Information Processing Society of Japan

```
#include <stddef.h>
#pragma commSemiring max "+" NINF 0
extern int NINF;
extern int max(int a, int b);

typedef struct BNint {
  int v;
  struct BNint *l, *r;
} BNint;
typedef struct pair_int {
  int _1, _2;
} pair_int;

pair_int mis(BNint *t) {
  pair_int ret, l = {NINF,0}, r = {NINF,0};
  if (t->l != NULL) l = mis(t->l);
  if (t->r != NULL) r = mis(t->r);
  ret._1 = t->v + l._2 + r._2;
  ret._2 = max(l._1, l._2) + max(r._1, r._2);
  return ret;
}
```

図 1　入力プログラムの例．2 分木上の最大独立和を計算する mis

Fig. 1　Example of input programs. mis computes the maximum independent sum of a given tree.

本システムでは，先行研究 [10] と同様に，reduce の計算しか扱わない．しかし，累積計算のスケルトン [8] についても，双線型計算における補演算子は同様の定式化で得られる．よって，入力プログラムの解釈を単純に拡張するだけで，累積計算を扱える．本システムが reduce に限定している理由は，実装コスト面だけにある．

3.2　本システムの利用例

図 1 と**図 2** に，本システムの典型的な利用例を示す．図 1 に示されるように，演算子生成器の入力は，単独でコンパイル可能な C プログラムである．我々が用いた演算子導出法では，可換半環について自由度がある．よって，#pragma を用いてユーザ定義の可換半環を宣言できる．演算子生成器の出力コードは C++であり，それを利用するコード（図 2）も C++であることを想定している．図 2 において，本質的な計算を行っているのは，binary_tree<int> 型のデータを構築する文と，そのデータに入口関数の parallel_dp::mis を適用する文の 2 つである．ここで，binary_tree<int> は，スケルトン専用の並列木データ構造の実装とする．利用者の視点では，逐次版のデータ型 BNint と関数 mis を，それぞれの並列版である binary_tree<int> と parallel_dp::mis に置き換えるだけで，並列計算が可能になる．

```
#include <skel/binary_tree.hpp>
#include <skel/config_main.hpp>
#include "mis-tree.cpp"

int main(int argc, char **argv)
{
  config::init(argc, argv);

  binary_tree<int> *t =
    binary_tree<int>::read_from_file("data");
  pair_int result = parallel_dp::mis(*t);

  config::finalize();
  return 0;
}
```

図 2　出力コードの利用例．mis-tree.cpp は，我々の演算子生成器に図 1 のプログラムを与えた結果の出力コード

Fig. 2　Example of use of output code. mis-tree.cpp is an output file that our operator generator generates, given the program shown in Fig. 1.

3.3　演算子生成用のディレクティブ

ここでは，我々の演算子生成器の API であるプリプロセッサディレクティブ（#pragma）を紹介する．

3.3.1　可換半環の宣言

我々の演算子生成器にとって，最も重要な API は，可換半環の宣言である．2 章で述べたように，我々が実装した演算子導出法は，可換半環上で一般化されているため，この代数構造をもとに演算子を導出する．

#pragma commSemiring plus times zero one という文法で可換半環 $(S, \texttt{plus}, \texttt{times}, \texttt{zero}, \texttt{one})$ を宣言する．ここで，可換半環の定義域 S は，明示的に宣言されない．これは，演算子が利用されている文脈から定義域（型）が判別可能だからである．このことで，組み込み演算子のようにオーバロードされる場合を簡潔に記述可能になり，単位元シンボルの暗黙の型変換が自然に行われるようになる．

組み込み演算子を，可換半環の演算子として使う場合は，文字列リテラルとして記述する（図 1）．組み込み演算子については，#pragma commSemiring "+" "*" 0 1 と #pragma commSemiring "|" "&" 0 1 の可換半環が，暗黙に定義済みとして扱われる．

我々は，宣言された可換半環の代数的性質を検査しない．これは，実装コストよりも，計算機上の標準的なデータ表現と演算では，代数的性質が厳密には成り立たないことに起因する．たとえば，算術オーバフローや丸め誤差の扱い，零元 $-\infty$ の表現は，厳格な代数的性質の実装を困難にする．よって，我々は，近似的な代数的性質を想定する．この「近似的」とは，たとえば，$-\infty$ を，計算過程で出現する値よりも十分に小さい値で代用するようなことである．

© 2013 Information Processing Society of Japan

3.3.2　純関数の宣言

2 章で述べたように，双線型計算における係数行列の成分は，可換半環上の 1 次式であることが要求される．この 1 次式の係数や定数項には，副作用をともなわない任意の関数を利用できる．要素値への述語（図 4 中の `all` と `any`）のような，木の要素型から演算の定義域へとリフトする関数は，アルゴリズム記述にほぼ必須である．よって，我々の演算子生成器は，`#pragma pure f` という構文で，関数 `f` が副作用のない純関数であると宣言する API を提供する．

宣言された可換半環の代数的性質と同様に，関数の純粋性を検査しない．これには，2 つ理由がある．第 1 に，定義が与えられない関数を扱うためである．そもそも，純関数宣言は，定義済みの関数を再利用するためにある．それをコンパイル単位に含めなければならなくなると，結局再定義しなければならず，再利用性が失われる．第 2 に，副作用を合理的に定義するのが難しいためである．一般に，副作用は，関数の返値以外の作用という意味で用いられるが，その意味の副作用を厳格に排除するのは，実行時環境を直接観測できる C/C++では非現実的である．たとえば，`sin` などの数学関数ですらも，errno を考えれば，副作用を持ってしまう．さらに，計算の本筋には関係しないエラーログ出力を，副作用と見なすべきかは議論が分かれるところである．そこで我々は，関数の純粋性の判断を，利用者に委ねることにした．

3.3.3　コード生成の抑制

我々の演算子生成器は，C ソースコードを受け取り，`#include` 用にカプセル化された C++ソースコードを生成する．C と C++ には，完全ではないものの，構文的な互換性があるため，入力の一部をそのまま出力することになる．これは，名前区間を不必要に汚すことになる．たとえば，図 1 における構造体 `BNint` は，関数 `mis` でしか使われず，デバッグを終えた `mis` は，それ自体が使われなくなりうる．このような不要なシンボルを，グローバルの名前空間に残さないようにするために，我々の演算子生成器は，コード生成を抑制する API を備える．`#pragma exclude name` の構文で，シンボル `name` のコード生成が抑制される．この機能を使うことで，入力では関数や変数であったものを，出力の利用側でマクロに置き換えるというトリッキーなことも可能になる．たとえば，図 5 の出力コードの利用側ならば，`max` と `NINF` をマクロで定義しても問題ない．

3.4　アルゴリズム記述の自由度と制約

本システムでは，アルゴリズム記述に制限を設定している．本節では，記述の自由度と制約について説明する．

3.4.1　アルゴリズム記述の自由度

図 1 の `mis` のような再帰関数は，典型的な 2 分木アルゴリズムの記述方法である．この記述形式以外にも，いくつかの再帰呼び出しの形式を，本システムは扱うことができる．

```
pair_int singleton(BNint *t) {
  pair_int ret, l = {0,0}, r = {0,0};
  if (t->l != NULL && t->r == NULL) {
    l = singleton(t->l);
    r._2 = 1;
  }
  if (t->l == NULL && t->r != NULL) {
    l._1 = 1;
    r = singleton(t->r);
  }
  if (t->l != NULL && t->r != NULL) {
    l = singleton(t->l);
    r = singleton(t->r);
  }
  ret._1 = l._1 + r._1;
  ret._2 = l._2 + r._2;
  return ret;
}
```

図 3　一人っ子を数える `singleton`

Fig. 3　`singleton`, which counts up the siblingless nodes in a given tree.

```
#pragma pure all any
extern int all(int);
extern int any(int);

pair_int left_lean(BNint *t) {
  pair_int ret, l = {1,0}, r = {1,0};
  if (t->l != NULL)  {
    l = left_lean(t->l);
    if (t->r != NULL) {
      r = left_lean(t->r);
    }
  }
  ret._1 = all(t->v) & l._2 & r._2;
  ret._2 = any(t->v) | l._1 | r._1;
  return ret;
}
```

図 4　左に傾いた走査をしつつ述語で要素値を調べる `left_lean`

Fig. 4　`left_lean`, which tests with two predicates the element values of a given tree in a *left-leaning* traversal.

たとえば，**図 3** に示される `singleton` は，一人っ子の数を数えるアルゴリズムである．これは，想定するノードの形について，1 つ 1 つ順にテストする形式である．

もう 1 つの走査例を**図 4** に示す．`left_lean` は，中間ノードがつねに左の子を持つことが想定された木の走査であり，左の子がなければ右の子の探索は行われない．

先行研究 [10] と同様に，全 2 分木として走査することも

© 2013 Information Processing Society of Japan

```
#pragma commSemiring max "+" NINF 0
#pragma exclude max NINF
extern int NINF;
extern int max(int a, int b);

pair_int mis_fbt(BNint *t) {
  pair_int ret;
  if (t->l == NULL && t->r == NULL) {
    ret._1 = NINF;
    ret._2 = 0;
  } else {
    pair_int l = mis_fbt(t->l),
             r = mis_fbt(t->r);
    ret._1 = t->v + l._2 + r._2;
    ret._2 = max(l._1,l._2) + max(r._1,r._2);
  }
  return ret;
}
```

図 5　全 2 分木の最大独立和を計算する mis_fbt
Fig. 5　mis_fbt, which computes the maximum independent sum of a given full binary tree.

できる．**図 5** に，全 2 分木上の走査の例を示す．この計算では，片側しか子が存在しない場合を考慮していない．このとき，本システムでは，片側しか子が存在しない場合を未定義とし，実行時例外を起こす未定義演算子を用いる．これによって，入力が全 2 分木ならば問題なく終了し，そうでない場合は実行時例外を起こすプログラムが得られる．

3.4.2　アルゴリズム記述の制約

並列化の対象関数への表現上の制約は，並列計算に由来するもの，演算子導出法に由来するもの，実装上の都合に由来するものの 3 つに分類できる．

並列計算に由来する制約は，副作用に関係するものである．純関数宣言のない関数の呼び出しや大域変数への書き込みが禁止されている．演算子導出法に由来する制約は，すなわち双線型計算に適合するため条件であり，再帰呼び出しやデータ型に関するものである．返値の構造体は，すべて同じ型で固定数のメンバで構成されなければならない．再帰呼び出しの結果を条件部に含む if 文，一度に 2 つ分ノードを下るような走査は扱えない．

実装上の都合に由来する制約は，4.1 節で述べるケース抽出や記号的実行の実装を容易にするためにある．たとえば，ポインタへの書き込みをしてはならない．2 分木のノード構造体は，要素値 v，左右の子へのポインタ l と r しか持ってはならない．関数は引数に NULL をとらず，定義には if 文以外の制御構造があってはならない．子のノードの存在確認は，NULL テストで行われることを前提としている．この実装上の都合による制約は，並列化と導出手法に強く関わるものもあり，記述対象のアルゴリズムが限定されているので，おおむね合理的な制約であると考える．

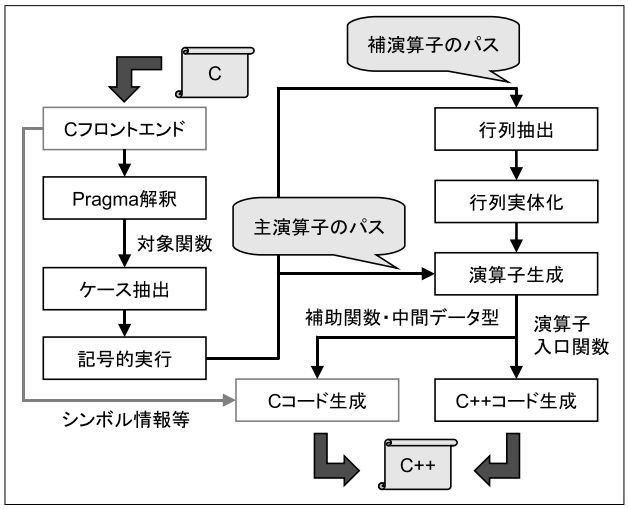

図 6　演算子生成器のコンパイルパイプライン
Fig. 6　Compilation pipeline of our operator generator.

4.　システムの実装概要

我々のシステムは，演算子生成器と 2 分木スケルトンライブラリの 2 つからなる．本章では，その実装の概要と処理の流れを説明する．

4.1　演算子生成器の実装概観

演算子生成器は COINS を用いて実装した．プログラムの変換は，入力された抽象構文木の構造をほぼ残した高水準中間表現 HIR [*4] 上で行った，演算子生成器のコンパイルパイプラインを**図 6** に示す．以降は，図 6 が示す処理の流れに従って，図 1 を入力例として説明する．

4.1.1　Pragma 解釈と前処理

C フロントエンドが構築した HIR から，#pragma を見つけて解釈する．#pragma の指定がない場合は，組み込みの可換半環を用いて，すべての関数定義に並列化を試みる．#pragma commSemiring によって，ユーザ定義関数が指定された場合は，その関数に 1 対 1 対応する一意な HIR 上の演算子番号を生成する．そして，関数呼び出しを 2 項演算子式へと内部表現を変換する．

図 1 のプログラムに含まれる #pragma を解釈することで，max 関数が演算子として扱われる．さらに，max と + が可換半環をなすことが仮定される．

4.1.2　ケース抽出

演算子生成器は，入力プログラム中のすべての関数を並列化対象の関数と扱う．まず，ケース抽出器が，主演算子の場合分けごとに関数定義をスライスする．具体的には，引数のノードが持つ左右の子へのポインタが，両方 NULL（E; End），右のみ NULL（L; Left），左のみ NULL（R; Right），両方 NULL でない（B; Branch）という 4 通

[*4] コンマ演算子や短絡評価論理演算子が除去され，代入が式ではなく文の扱いになるなど，扱いやすく正規化されている．

© 2013 Information Processing Society of Japan

りのスライスを生成する．

以降では，図 1 の mis 関数を例に用いて説明する．mis において右のみ NULL の場合のスライスは，次のようになる．

```
pair_int mis(BNint *t) {
  pair_int ret, l = {NINF,0}, r = {NINF,0};
  l = mis(t->l);
  ret._1 = t->v + l._2 + r._2;
  ret._2 = max(l._1, l._2) + max(r._1, r._2);
  return ret;
}
```

注意すべき点は，COINS の C フロントエンドが，短絡評価の論理演算子 && と || を，goto をともなった if 文へと展開することである．本実装では，この goto を，then/else 部の複製によって前もって除去し，ケース抽出を簡単化する．

4.1.3　記号的実行

次に，それぞれのスライスについて，記号的実行を行い，関数定義を単独の式へと変形させる．具体的には，出現するすべての式に副作用がないと仮定して，式の複製を作りつつ環境を更新しながら抽象実行し，return される式を記号的定数と引数のみの形にする．ノードの要素値と左右の子へ再帰呼び出しの 3 つは，それぞれ一意の変数で抽象化される．これで，各主演算子の定義と等価な式が得られる．それらを e_E，e_L，e_R，e_B と呼ぶ．

たとえば，上記のスライス上で記号的実行を行うと，e_L として次の式を得る．

```
_pair_int(_v + _l._2 + 0,
          max(_l._1, _l._2) + max(NINF, 0))
```

ここで，_pair_int は演算子生成器が便宜上作る pair_int の構築関数である．_v と _l は，それぞれ，ノードの要素値と左の子への再帰呼び出しの結果を抽象化する変数である．

e_E，e_L，e_R，e_B を得た後，一度その式中の変数出現を確認し，妥当性を確かめる．未定義の局所変数や，存在しない子に対する再帰呼び出しの結果を表す変数がある場合，そのケースの演算子が未定義であったと判断し，専用のパスで未定義演算子を生成する．

4.1.4　行列抽出

補演算子を導出するには，ボトムアップ演算を双線型計算として定式化しなければならない．具体的には，中間ノードに対する演算子を，行列ベクトル積の形に変形すればよい．線型性を利用することで，単に式を微分するだけで係数行列を抽出できる．微分の過程で非定数の導関数が出現したら，ただちに抽出失敗とすることで，線型性の検査も兼ねることができる．

e_B からは，r について微分して得られる A_L と，l について微分して得られる A_R，の 2 つの係数行列が抽出される．これらは 2 章の説明どおり，それぞれ ψ_L と ψ_R の元となる．e_R からは A'_L が，e_L からは A'_R が抽出される．それぞれ，ψ_L と ψ_R において葉ノードである子が存在しない場合の演算子 ψ'_L と ψ'_R の元となる．

mis の e_L からは，次の A'_R が抽出される．

$$\begin{pmatrix} \mathtt{NINF} & \mathtt{\_v} & \mathtt{NINF} \\ 0 & 0 & \mathtt{NINF} \\ \mathtt{NINF} & \mathtt{NINF} & 0 \end{pmatrix}$$

ここで，$(1,2)$-成分は，e_L の第 1 メンバの式 _v + _l._2 + 0 を，_l の第 2 メンバ _l._2 で微分した結果に対応する．

微分する過程で，対象の可換半環に基づいた定数伝搬も行う．これは，出力コードの実行時性能を向上させる効果もあるが，後の行列実体化において，抽象行列の構築を簡単にする役割もある．

4.1.5　行列実体化

抽出された行列は，たいてい（可換半環上の）0 と 1 の成分を多く持っており，その一部は計算中不変になる場合がある．その不変な 0 と 1 の成分は，実行時データとして保持する必要はなく，コード生成時に演算子の中に埋め込むだけでよい．行列実体化器は，不変な 0 と 1 の成分を見つける．2 章で述べたように，縮約演算でのみ行列が更新されるので，抽出された行列での行列連鎖積を抽象実行すればよい．各行列成分を変数にする必要があるかどうかさえ分かれば十分なので，抽象実行に用いられる抽象行列の成分値は，0 と 1 と変数 V の 3 値であればよい．その 3 値行列に合わせて，抽象行列乗算と抽象行列更新を定義する．この抽象実行の不動点となる 3 値行列から，特化した具象行列のデータ型は生成できる．抽象実行の詳細は，Matsuzaki [8] を参照されたい．

mis における A_L, A_R, A'_L, A'_R の抽象行列からなる行列連鎖積の不動点は，次のようになる．

$$\begin{pmatrix} V & V & 0 \\ V & V & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

よって，具象行列は V で示された 2×2 行列で十分である．ここから，次のような具象行列を表す構造体 _mis_mat が作られる．

```
struct _mis_mat {
 int  _0_0, _0_1, _1_0, _1_1;
};
```

構造体のメンバ名は，0 開始の行列成分の添字に対応する．

4.1.6　演算子生成とコード生成

コード生成は，2 章で述べたとおりで，直截である．主演算子については，e_E，e_L，e_R，e_B に return 文を加えて，適切な関数宣言構築するだけである．補演算子については，A_L，A_R，A'_L，A'_R に対する記号的な行列乗算を本体

© 2013 Information Processing Society of Japan

として，適切な関数宣言を構築するだけである．抽象化のために導入された変数は，ここで演算子の仮引数に置き換えられる．

mis の A'_R からは，次のような ψ'_R に対応する関数（関数オブジェクトのメンバ関数）が生成される．

```
hir_t_void   operator()(const  hir_t_int
v, struct  _mis_mat  &m) const
{
  hir_s_private hir_v_auto hir_t_int  _var43;
  hir_s_private hir_v_auto hir_t_int  _var45;
  hir_s_private hir_v_auto hir_t_int  _var47;
  hir_s_private hir_v_auto hir_t_int  _var49;
  _var43 = (hir___ADD(v,m._1_0));
  _var45 = (hir___ADD(v,m._1_1));
  _var47 = max( m._0_0,m._1_0);
  _var49 = max( m._0_1,m._1_1);
  m._0_0 = _var43;
  m._0_1 = _var45;
  m._1_0 = _var47;
  m._1_1 = _var49;
}
```

コード生成の直前で，出力コードに含めたくないシンボル情報を除去する．たとえば，wchar_t は C では typedef された型だが，C++では予約語であり，typedef するとエラーになる．よって，wchar_t の typedef は，出力を抑制される．同様に，#pragma exclude で指定されたシンボル情報も削除する．

4.2　2 分木スケルトンライブラリ

本システムが入力とする手続き的アルゴリズム記述においては，片方の子が存在しない場合の計算が自然に生じる．そういう場合を対処できるように，Matsuzaki [8] が実験的に実装していた，全 2 分木スケルトンライブラリをもとに，データ構造と reduce の実装を書き直す形で，2 分木スケルトンライブラリを新たに実装した．2 分木スケルトン reduce が要求する，演算子の代数的性質は，付録 A.1 を参照されたい．reduce の入力となる並列木データ構造は，m-bridge [14] の技法に基づいて，部分木と単穴文脈木に分割し，ノードを行きがけ順で並べて配列に保持する実装法をとった．本ライブラリは，既存ライブラリと同様に，MPI [*5]環境での並列実行を想定している．実際，図 2 における関数 config::init と config::finzalize は，MPI 関数のラッパとなっている．

入口関数は，複数のスケルトン実装を統一して扱える多相関数として実装すると，演算子生成器の出力コードの移植性が高まる．それを実現するために，データ構造からスケルトンの実装を共通した API によって呼び出せるように設計した．この点に関する既存ライブラリとの実装上の差異は些細だが，設計上は注目すべき点である．

[*5] http://www.mpi-forum.org/

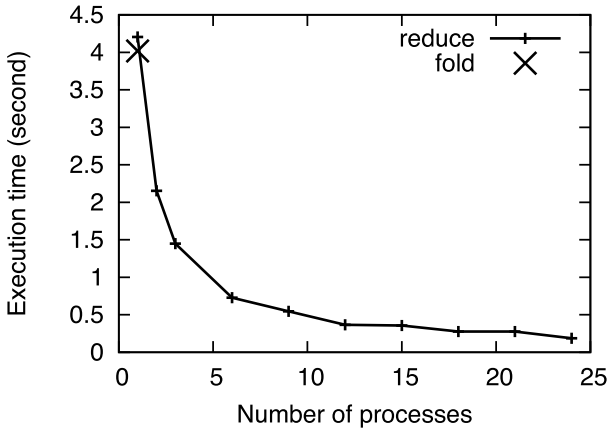

図 7　mis の計算における reduce と fold の実行時間．木のノード数は 2^{28}．木の分割数は 48．木の構築時間は含まれない

Fig. 7　Execution time of reduce and fold for mis. The number of input nodes is 2^{28}; the number of tree segments is 48. Time for tree construction is not contained.

5.　動作実験

図 2 と同等のプログラムを用いて，動作実験を行った．動作環境は，1000BASE-T で接続された 3 ノードの PC クラスタである．各ノードは，AMD Opteron 2376（2.3 GHz，4 コア，2 ソケット）と 8 GB メモリからなるサーバであり，Linux 3.2.0（64-bit）が動作していた．バイナリ生成には g++ 4.6.3 を用いて，O3 の最適化を施した．OpenMPI 1.4.3 を用いた．入力として，ランダム生成された 2^{28} ノードの木を用いた．実行時性能を理解しやすくするために，分割された断片の木は，どれもほぼ同じノード数[*6]となるように調整し，分割数を 48 とした．fold と reduce の実行時間を図 7 に示す．

結果は，ほぼ期待どおりの台数効果を得ることができた．1 プロセスの reduce が fold よりも遅い原因は，縮約演算にある．双線型計算における縮約演算は，本来なら行列ベクトル積で計算できる部分を，行列行列積によって計算するために，本質的にオーバヘッドを導入する．このオーバヘッドの大きさは，行列サイズと入力木の高さに依存する．今回観測されたオーバヘッドは，許容される範囲の小さいものであったが，この結果に一般性はない．

6.　多分木への応用

Matsuzaki [8] の木スケルトンは，全 2 分木上で定式化されている．一方，本研究では，4.2 節で述べたように，一般の 2 分木上で定式化された木スケルトンを用いた．これは，我々のシステムの利用者の立場からすれば，大した違いではないが，ライブラリ実装者の立場からは，大きな違いで

[*6] 一般に，m-bridge を用いて木を分割した場合，断片の大きさには，m に応じてある程度ばらつきが生じる．m の選び方と，計算量への影響については，文献 [8] に詳しく述べられている．

© 2013 Information Processing Society of Japan

表 1　木の次数と reduce の演算子の関係．表中の値は個数
Table 1　Relation between the arity of input trees and the parameter operators of reduce. Each value means the number of ones indicated in the leftmost column.

	全 2 分木	2 分木	k 分木
NULL テスト	1	2	k
ノードの種類	2	4	2^k
主演算子	2	4	2^k
補演算子	4	8	$2^{k-1}(k+2)$
演算子の制約	3	15	4^k-1

ある．なぜなら，木のノードの種類（代数的データ型ではデータ構成子）が増えると，木スケルトンが要求する演算子の数が爆発するからである．実際，付録 A.1 に示される全 2 分木 reduce と 2 分木 reduce は，API の複雑さが大きく異なる．演算子数の違いも大きいが，補演算子の代数的性質，つまり演算子の制約の数も大きく増えることに注目してほしい．我々が利用した 2 分木 reduce は，Matsuzaki [8] の定式化を，素朴に拡張することで得られる．同様に拡張することで，k 分木 reduce も得られる．**表 1** に示されるように，それらの reduce を比較すると，API の複雑さが，木の次数 k に対して指数的にふくらんでいることが分かる．特に，演算子の制約は，最も急速に複雑化する．

ここで，表 1 に示される NULL テストの項目に注目してほしい．これは，我々のシステムを使って，素朴な再帰関数で定義した場合に必要な NULL テスト，すなわち if 文の数である．NULL テストは当然，次数と同数しか必要ない．だからこそ，我々のシステムの利用者にとっては，全 2 分木か 2 分木かは，些細な問題なのである．別の言い方をすれば，我々のシステム設計は，木の次数に由来するスケルトンの複雑さに対して，堅牢であり，利用者に負担を強いない．したがって，多分木スケルトンを扱う場合には，我々のシステムの重要性は，さらに大きくなる．

7. 関連研究

これまで，リスト上の計算に対する演算子自動導出は，いくつか研究されている [13], [15], [18], [19]．この中で，我々のシステムと設計上最も近いのは，Xu らの PType システム [18] である．これは，ユーザ定義の半環宣言をもとにして，型推論によって線型式であることを保証する．そして，型の付いた項から，リストスケルトンの一般形であるリスト準同型の定義を生成する．彼らが用いた型解析と型主導プログラム変換は，我々の行列抽出と実質的に等しい．限量子除去に基づく導出 [12] は，理論的には木計算にも適用可能である．しかし，実装はリスト上の計算にしか適用されておらず，木計算に対して現実的に実行可能であるかは未知数である．

木上の並列性[*7]を扱う並列プログラミング言語としては，NESL [3] があげられる．これは，内包表記の形で表現されるリストスケルトンと純粋な再帰関数によって，暗黙裡に並列アルゴリズムを記述する枠組みである．NESL では，平坦化 [2] と呼ばれる技法によって，木上の並列性が，リスト上の並列性に変換されて，最終的にベクタ操作として実行される．この平坦化は，既存の言語へ導入するときに問題になる [1], [7]．詳細はそれらの論文に任せるが，問題を端的にいえば，平坦化されたデータ表現自体の空間的オーバヘッドと，そのデータ表現上の操作の時間的オーバヘッドである．この問題の根本的な原因は，第 1 に，木上の並列計算を直接扱っていないこと，第 2 に，木構造の抽象化が脆弱であることにある．基本的に NESL では，無制限の再帰を許可し，木の構築はリスト構築の再帰的反復で行う．木上の計算も同様に，リスト計算の再帰的反復で行う．つまり NESL では，木上の並列計算自体が定式化されておらず，リスト計算の合成しか存在しない．このために，平坦化によってリスト計算で定式化しない限り，NESL では木上の並列計算が不可能である．さらに，木を木として定式化していないために，データ構造の抽象化が不十分となり，リスト操作を隠蔽することができない．これでは，木計算に特化したデータ構造の実装を導入できず，平坦化によるリスト表現以外に選択肢がない．一方，木スケルトンは，木計算自体を定式化したものであり，木構造を抽象化する役割もある．そのため木スケルトンは，ライブラリ実装の形で，既存の言語へと自然に埋め込むことができる．

Matsuzaki [8] による木スケルトン定式化の基礎は，Tree Contraction アルゴリズム [14] である．近年の Morihata の研究 [11], [20] により，このアルゴリズムの枠組みで，属性文法やマクロ木変換器の評価を効果的に扱えることが分かった．しかし，それらはあくまで理論的成果であり，木スケルトンでの実行可能性や実用性は未知数である．

8. まとめ

本研究では，既存手法 [10] に基づき，木上のスケルトン並列プログラミングシステムを実装した．本システムが木スケルトン自体を隠蔽することで，利用者は木スケルトンの複雑さに触れることなく，木上の逐次アルゴリズムを記述するだけで，スケルトン並列プログラミングが行える．

利用者の与えたアルゴリズム記述から，適切な木スケルトンの実装を選択し合成するシステムは有益である．Boost[*8]のような C++テンプレートメタプログラミングに基づくライブラリと協調することで，実装の選択と合成をライブラリ側に移譲する設計が可能になる [9]．この設計は，演算子生成器の簡潔性と，出力コードの拡張性の点で有望である．

一方，木スケルトン自体が，実用に向けて多くの課題を残している．7 章で述べたように，理論的には木の変形を

*7　NESL の文脈では Nested Data Parallelism と呼ばれる．
*8　http://www.boost.org/

© 2013 Information Processing Society of Japan

扱えるが，木スケルトンはデータ構造の抽象化を守らなければならないので，演算子にデータ構築子を単に含めることは許可できない．さらに，全域的計算しか提供しないため，部分的な木計算を繰り返すパターンは，効率良く扱えない．したがって，演算子生成器とともに，木スケルトン自体を設計することが，重要な課題である．

謝辞　本研究を支援していただいた岩崎英哉教授，本研究に有益な批評をくださった森畑明昌助教に感謝いたします．

参考文献

[1] Bergstrom, L., Fluet, M., Rainey, M., Reppy, J., Rosen, S. and Shaw, A.: Data-Only Flattening for Nested Data Parallelism, *Proc. PPoPP '13*, pp.81–92 (2013).

[2] Blelloch, G.E.: *Vector Models for Data-Parallel Computing*, The MIT Press (1990).

[3] Blelloch, G.E., Chatterjee, S., Hardwick, J.C., Sipelstein, J. and Zagha, M.: Implementation of a Portable Nested Data-Parallel Language, *J. Parallel Distrib. Comput.*, Vol.21, No.1, pp.4–14 (1994).

[4] Cole, M.: *Algorithmic Skeletons: Structured Management of Parallel Computation*, Research Monographs in Parallel and Distributed Computing, MIT Press (1989).

[5] Dijkstra, E.W.: Letters to the Editor: Go To Statement Considered Harmful, *Comm. ACM*, Vol.11, No.3, pp.147–148 (1968).

[6] Gibbons, J., Cai, W. and Skillicorn, D.B.: Efficient Parallel Algorithms for Tree Accumulations, *Sci. Comput. Program.*, Vol.23, No.1, pp.1–18 (1994).

[7] Keller, G., Chakravarty, M.M.T., Leshchinskiy, R., Lippmeier, B. and Peyton Jones, S.: Vectorisation Avoidance, *Proc. Haskell '12*, pp.37–48 (2012).

[8] Matsuzaki, K.: Parallel Programming with Tree Skeletons, Ph.D. Thesis, The University of Tokyo (2008).

[9] Matsuzaki, K. and Emoto, K.: Implementing Fusion-Equipped Parallel Skeletons by Expression Templates, *Proc. IFL '09*, LNCS, Vol.6041, pp.72–89 (2010).

[10] Matsuzaki, K., Hu, Z. and Takeichi, M.: Towards Automatic Parallelization of Tree Reductions in Dynamic Programming, *Proc. SPAA '06*, ACM, pp.39–48 (2006).

[11] Morihata, A.: Macro Tree Transformations of Linear Size Increase Achieve Cost-Optimal Parallelism, *Proc. APLAS '11*, LNCS, Vol.7078, pp.204–219 (2011).

[12] Morihata, A. and Matsuzaki, K.: Automatic Parallelization of Recursive Functions using Quantifier Elimination, *Proc. FLOPS '10*, LNCS, No.6009, pp.321–336 (2010).

[13] Morita, K., Morihata, A., Matsuzaki, K., Hu, Z. and Takeichi, M.: Automatic Inversion Generates Divide-and-Conquer Parallel Programs, *Proc. PLDI '07*, pp.146–155 (2007).

[14] Reif, J.H. (Ed.): *Synthesis of Parallel Algorithms*, chapter 3 List Ranking and Parallel Tree Contraction, Morgan Kaufmann Publishers Inc. (1993).

[15] Sato, S. and Iwasaki, H.: Automatic Parallelization via Matrix Multiplication, *Proc. PLDI '11*, pp.470–479 (2011).

[16] Skillicorn, D.B.: The Bird-Meertens Formalism as a Parallel Model, *Software for Parallel Computation*, NATO ASI Series, Vol.106, pp.120–133, Springer (1993).

[17] Skillicorn, D.B.: Parallel Implementation of Tree Skeletons, *J. Parallel Distrib. Comput.*, Vol.39, No.2, pp.115–125 (1996).

[18] Xu, D.N., Khoo, S.-C. and Hu, Z.: PType System: A Featherweight Parallelizability Detector, *Proc. APLAS '04*, LNCS, Vol.3302, pp.197–212 (2004).

[19] 松崎公紀，胡　振江，武市正人：リスト上の最大マーク付け問題を解く並列プログラムの導出，情報処理学会論文誌：プログラミング，Vol.49, No.3 (PRO36), pp.16–27 (2008).

[20] 森畑明昌：並列木縮約を用いたマクロ木変換器の並列評価，日本ソフトウェア科学会第 29 回大会講演論文集 (2012).

付　　録

A.1　補演算子の代数的性質

本質的な部分の理解のために，まず全 2 分木の場合について述べる．以降では，Haskell[*9]記法を用いる．全 2 分木を表す *BTree* のデータ型定義と，その上の fold は次のように定義される．

$$\begin{aligned}&\textbf{data}\ \mathit{BTree}\ a = \mathit{Branch}\ (\mathit{BTree}\ a)\ a\ (\mathit{BTree}\ a) \mid \mathit{Leaf}\ a\\&\text{fold}\ k_E\ k_B\ (\mathit{Branch}\ l\ v\ r)\\&\qquad = k_B(\text{fold}\ k_E\ k_B\ l)\ v\ (\text{fold}\ k_E\ k_B\ r)\\&\text{fold}\ k_E\ k_B\ (\mathit{Leaf}\ v) = k_E\ v\end{aligned}$$

BTree 上の reduce の型と，定義の左辺は次のようになる．

$$\begin{aligned}&\mathsf{reduce} :: (a \to b) \to (b \to a \to b \to b) \to (a \to c)\\&\qquad \to (b \to c \to b \to b) \to (b \to c \to c \to c)\\&\qquad \to (c \to c \to b \to c) \to \mathit{BTree}\ a \to b\\&\mathsf{reduce}\ k_E\ k_B\ \tau\ \phi_B\ \psi_L\ \psi_R\ t = \ldots\end{aligned}$$

定義の右辺は，具体的な実装に依存するが，数学的には $\text{fold}\ k_E\ k_B\ t$ と同等の結果を返す．この reduce の補演算子の代数的性質は，次の 3 つである．

$$\begin{aligned}k_B\ x\ l\ r &= \phi_B\ l\ (\tau\ x)\ r\\\phi_B\ l\ x\ (\phi_B\ l'\ x'\ r')\ &= \phi_B\ l'\ (\psi_L\ l\ x\ x')\ r'\\\phi_B\ (\phi_B\ l'\ x'\ r')\ x\ r\ &= \phi_B\ l'\ (\psi_R\ x'\ x\ r)\ r'\end{aligned}$$

一般の 2 分木を表す *GBTree* は，次のように定義される．

$$\begin{aligned}\textbf{data}\ \mathit{GBTree}\ a = {}&\mathit{Branch}\ (\mathit{GBTree}\ a)\ a\ (\mathit{GBTree}\ a)\\&\mid \mathit{Left}\ (\mathit{GBTree}\ a)\ a\\&\mid \mathit{Right}\ a\ (\mathit{GBTree}\ a)\\&\mid \mathit{Leaf}\ a\end{aligned}$$

ここで，*Left* は左の子のみ持つノード，*Right* は右の子のみ持つノードを表す．*GBTree* 上の fold も，全 2 分木と同様に，各データ構成子を対応する演算子で置換して評価する計算として定義される．ここで，k_E は *Leaf* に，k_L は *Left* に，k_R は *Right* に，k_B は *Branch* に対応する．*GBTree* 上の reduce の型と定義の左辺は，次のようになる．

[*9] http://www.haskell.org/

© 2013 Information Processing Society of Japan

$$
\begin{aligned}
&\mathsf{reduce} :: (a \to b) \to (a \to b \to b) \to (a \to b \to b) \\
&\qquad \to (b \to a \to b \to b) \to (a \to c) \\
&\qquad \to (b \to c \to b \to b) \to (b \to c \to c \to c) \\
&\qquad \to (c \to c \to b \to c) \to (c \to c \to c) \\
&\qquad \to (c \to c \to c) \to \mathit{GBTree}\ a \to b \\
&\mathsf{reduce}\ k_E\ k_L\ k_R\ k_B\ \tau\ \phi_L\ \phi_R\ \phi_B\ \psi_L\ \psi_R\ \psi'_L\ \psi'_R\ t = \ldots
\end{aligned}
$$

全 2 分木の場合と同様に，定義の右辺は，数学的には $\mathrm{fold}\ k_E\ k_L\ k_R\ k_B\ t$ と同等の結果を返す．この補演算子の代数的性質は次のようになる．

$$
\begin{aligned}
k_B\ x\ l\ r &= \phi_B\ l\ (\tau\ x)\ r \\
\phi_B\ l\ x\ (\phi_B\ l'\ x'\ r')\ &= \phi_B\ l'\ (\psi_L\ l\ x\ x')\ r' \\
\phi_B\ l\ x\ (\phi_L\ l'\ x')\ &= \phi_L\ l'\ (\psi_L\ l\ x\ x') \\
\phi_B\ l\ x\ (\phi_R\ x'\ r')\ &= \phi_R\ (\psi_L\ l\ x\ x')\ r' \\
\phi_B\ (\phi_B\ l'\ x'\ r')\ x\ r &= \phi_B\ l'\ (\psi_R\ x'\ x\ r)\ r' \\
\phi_B\ (\phi_L\ l'\ x')\ x\ r &= \phi_L\ l'\ (\psi_R\ x'\ x\ r) \\
\phi_B\ (\phi_R\ x'\ r')\ x\ r &= \phi_R\ (\psi_R\ x'\ x\ r)\ r' \\
k_L\ l\ x &= \phi_L\ l\ (\tau\ x) \\
\phi_L\ (\phi_B\ l'\ x'\ l')\ x &= \phi_B\ l'\ (\psi'_L\ x\ x')\ r' \\
\phi_L\ (\phi_L\ l'\ x')\ x &= \phi_L\ l'\ (\psi'_L\ x\ x') \\
\phi_L\ (\phi_R\ x'\ r')\ x &= \phi_R\ (\psi'_L\ x\ x')\ r' \\
k_R\ x\ r' &= \phi_R\ (\tau\ x)\ r' \\
\phi_R\ x\ (\phi_B\ l' x'\ r')\ &= \phi_B\ l'\ (\psi'_R\ x'\ x)\ r' \\
\phi_R\ x\ (\phi_L\ l'\ x')\ &= \phi_L\ l'\ (\psi'_R\ x'\ x) \\
\phi_R\ x\ (\phi_R\ x'\ r')\ &= \phi_R\ (\psi'_R\ x'\ x)\ r'
\end{aligned}
$$

佐藤 重幸

電気通信大学大学院情報理工学研究科情報・通信工学専攻博士後期課程に現在在籍中．並列プログラミング，プログラム変換，ドメイン特化言語，コンパイラ等に興味を持つ．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，2009 年より高知工科大学情報学群准教授となり現在に至る．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，ACM，IEEE 各会員．

© 2013 Information Processing Society of Japan