2012-5-(6): 情報処理学会プログラミング研究会 発表資料 2013 年 2 月 28 日

boost::proto を用いた融合変換機能付きライブラリの作成

松崎 公紀[1,a)]　江本 健斗[2]

概要：融合変換は, 複数の関数や繰り返しを 1 つにまとめることにより中間データを除去するプログラム変換である. 小さな部品を組み合わせるようなプログラミングスタイルにおいて, 融合変換を行うことで中間データを除去することにより, プログラムの効率を向上させることができる. C++ 言語においてこの融合変換を実現する手法の 1 つに, 式テンプレートを用いたメタプログラミング手法がある. しかし, 従来の式テンプレートを直接記述するメタプログラミングでは, 式から抽象構文木を構成する部分と最適化を行う部分が混在してしまうため, プログラムが複雑になってしまう. 近年, 式テンプレートを用いたプログラムを記述しやくするライブラリとして boost::proto が作られ, 利用できるようになった. boost::proto を用いることで, 融合変換などの最適化機能付きライブラリを容易にかつ見通し良く作成できることが期待される. 著者らは, 並列計算における計算パターン (並列スケルトン) を提供する並列スケルトンライブラリを作成している. その中で, 並列スケルトンの融合変換機能を boost::proto を用いて, より高い拡張性を持つように書き換えたいと考えている. 本論文では, 融合変換機能を持つライブラリを boost::proto を用いて作成することで得られる利点や問題点などについて報告する.

キーワード：テンプレートメタプログラミング, 融合変換, boost::proto

Implementing a Fusion-equipped Library with boost::proto

Kiminori Matsuzaki[1,a)]　Kento Emoto[2]

Abstract: Fusion transformation is a program transformation that composes functions or loops into a single one to remove intermediate data. In a programming style in which we compose small program pieces, we can gain the performance of a program with fusion transformation. An approach for implementing fusion transformation in C++ is to use a meta-programming technique called expression templates. However, if we develop programs directly with expression templates, programs are often complicated due to mixed program pieces for constructing abstract syntax trees and for applying optimization to them. Recently, a template library called boost::proto has been developed for clear implementation of expression templates. We expect that we can develop a fusion-equipped library easily and clearly by using boost::proto. We have been developing a parallel skeleton library that provides computation patterns for parallel programming (called parallel skeletons). We try to rewrite the fusion optimization mechanism in the library using boost::proto so that it has high extendability. In this paper, we report the advantages and problems in the implementation of a fusion-equipped library with boost::proto.

Keywords: Template meta programming, fusion transformation, boost::proto

1. はじめに

多くのプログラミング環境で, さまざまなライブラリが作られ, また利用されている. その中でも数値計算などの分野では特に, 記述性だけでなく実行速度は重要な性質である. 高い記述性を持ちながら実行速度も十分に速いライブラリが最も望ましいものである.

[1] 高知工科大学 情報学群
School of Information, Kochi University of Technology, Kami, Kochi, 782–8502, Japan
[2] 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
[a)] matsuzaki.kiminori@kochi-tech.ac.jp

プログラム変換は, ユーザにとっての記述性を確保しつつ, 実行速度が速いライブラリを作ることができる有効なアプローチである. ユーザは明瞭な仕様を記述し, 正しさの保証されたプログラム変換を適用することにより, 実行速度の速いプログラムを得る. 比較的小さな部品を組み合わせることでプログラムを開発する場合には, 複数の関数や繰り返しをまとめて中間データを除去する融合変換 [7], [8], [16] は特に有益である. したがって, 自身が融合変換機能を持つようなライブラリの開発は重要な研究テーマである.

C++ 言語においてこの融合変換を実現する手法の 1 つに, 式テンプレートを用いたメタプログラミング手法がある. この式テンプレート手法による高性能なライブラリの開発も行われている [5], [6], [9], [10], [17]. しかし, 従来の式テンプレートを直接記述するメタプログラミングでは, 式から抽象構文木を構成する部分と最適化を行う部分が混在してしまうため, プログラムが複雑になってしまうことが問題であった. 近年, 式テンプレートを用いたドメイン特化組込言語を開発しやすくするライブラリ boost::proto [13] が作られ, 利用できるようになった. boost::proto を用いることで, 融合変換などの最適化機能付きライブラリを容易にかつ見通し良く作成できることが期待される.

著者らは, 並列計算における計算パターン (並列スケルトン) を提供する並列スケルトンライブラリを開発している [10]. これまでは, 最適化機能を式テンプレートを用いて直接記述していたが, その拡張性や保守性が課題となっていた. そこで, boost::proto を用いることで, 高い拡張性を持つように並列スケルトンの融合変換機能を再実装することを検討した. 本論文では, boost::proto を用いて最適化機能をどのように実装できるかについて調査し, 最適化機能付きライブラリの開発への利点と問題点を議論することを目標とする.

本論文の貢献は次の 2 点である.

- 簡単な融合変換である map-map 融合を例題として, 最適化手法が boost::proto を用いて 2 通りの方法で実現できることを示す.
- boost::proto を用いて最適化機能を持つライブラリを実現するにあたって, その利点と問題点を議論する. boost::proto を用いると, これまで著者らが実現していない最適化を含めることが可能そうであるという知見を得た.

本論文の構成は以下の通りである. 第 2 章では, 基礎知識として C++のテンプレートおよび式テンプレートについて述べた後, boost::proto を紹介する. 第 3 章では, map-map 融合を例題として, boost::proto を用いてどのように最適化機能付きライブラリを実現できるかについて述べる. 第 4 章では, 第 3 章の結果を受けて, 最適化機能付きライブラリの開発に boost::proto を用いる利点と問題点, さらに関連研究について議論する. 第 5 章で本論文をまとめる.

2. 式テンプレートと boost::proto

2.1 テンプレート

C++におけるテンプレートは, 型 (もしくは整数定数) をパラメータとする汎用的な関数や構造体を作る仕組みである. 例えば, その要素の型が `A` であるような長さ `SIZE` の配列をラップする構造体を,

```
template <typename A>
struct array {
  A data[SIZE];
};
```

のように定義すると, `array<int>` によって `int` 型の配列の構造体として, また, `array<double>` によって `double` 型の配列の構造体として利用できる.

C++のテンプレートはコンパイル時に静的に展開される. テンプレートを用いて定義された関数や構造体は, コンパイル時にその 1 つのインスタンスごとに 1 つのコードが生成される. したがって, あるプログラム中で `array<int>` と `array<double>` が使われている場合には, それぞれに対応する構造体のコードが生成される.

C++のテンプレートがコンパイル時に静的に展開されることを利用すると, コンパイル時にある種の計算を行うことができる. この, テンプレートを用いてコンパイル時に計算を行わせるプログラミングは, テンプレートメタプログラミングと呼ばれる. テンプレートメタプログラミングでは, 型 (もしくは整数定数) を使って計算を行う. それらの型 (や整数) はコンパイル時に静的に決定されるため, さらにコンパイラの最適化を期待することができる.

C++において, 関数の引数として式を渡すには, 関数へのポインタもしくは関数オブジェクトを用いる. このうち, 関数オブジェクトを用いると, コンパイル時にインライン展開して最適化することにより高速なコード生成が期待できる. 例えば, 図 1 の `map` 関数は, `const F` 型の関数オブジェクト `f` をとり, その関数を配列のすべての要素に適用するものである. GCC などのコンパイラで最適化を有効にしてコンパイルすると, この関数オブジェクトの関数呼び出しはインライン展開され, 繰り返しを直接記述するのと同じコードが生成される.

2.2 式テンプレート

式テンプレート (expression template) は, テンプレートメタプログラミングの手法の 1 つである. そのアイデアは, (数学的な) 式をテンプレートによって抽象化された関数オブジェクト (関数 `operator()` を含む構造体) として構成し, それを用いて計算を行うというものである. これにより, 関数の評価を遅延させたり, プログラムを変換することができる.

式テンプレートの例として, map 関数に値を 10 倍する関数を渡す例を示す.

```
struct Var {
  int operator()(int v) { return v; }
};
template <typename X>
struct Mult {
  int a; X x;
  Mult(a_, x_) : a(a_), x(x_) {};
  int operator()(int v){return a * x(v);}
};

...
  map(Mult<Var>(10, Var()), xs);
```

上のプログラムにおいて, map 関数の第 1 引数として Mult<Var> 型のオブジェクトが渡される. この Mult<Var> 型は, 式 $\lambda x.a * x$ をテンプレートによって表現したものとなっている. コンパイル時に展開されることにより, その Mult<Var> 型に特化した map 関数が作られ, 最適化の後に

```
for (int i = 0; i < size; i++) {
    ret[i] = 10 * xs[i]; }
```

というプログラムと等価なコードが生成される.

Var などのオブジェクトを引数にとり式全体を表すオブジェクトを生成するように演算子をオーバーロードすると, 性能を犠牲にすることなくプログラムをより読み易くすることができる. 定数を表す構造体 Con と, 積を表すオブジェクトを生成する演算子 * を用いて, 次のようなプログラムを書くこともできる.

```
map(Con(10) * Var(), xs);
```

2.3 boost::proto

Niebler によって作られた boost::proto [13] は, ドメイン特化組込言語 (domain-specific embedded language) を容易に扱えるようにすることを目的としたテンプレートライブラリである. boost ライブラリのうち, 関数プログラミングをサポートする Phoenix [3] や式テンプレートとして書ける正規表現ライブラリ Xpressive [12] などは, この boost::proto を用いて構築されている.

boost::proto において提供される機能は大きく次の 4 つである.

- C++ の文法に違反しない式から抽象構文木を構築する.
- ドメイン特化組込言語の文法を定義し, 抽象構文木がその文法を満たすかどうかを判定する.
- 抽象構文木に対してその評価を行う文脈 (context) を定義する.
- 抽象構文木を変換 (transform) する.

boost::proto では, C++の演算子をオーバーロードしているため, 抽象構文木の種を含む (演算子による) 式から抽象構文木が生成される. この抽象構文木は, 通常の C++プログラムの意味とは独立となっている. ドメイン特化組込言語の開発者は, 作られた抽象構文木が従うべき文法とともに抽象構文木から計算を行う文脈 (context) を定義する. また, 抽象構文木をそのまま評価するだけでなく, 評価の前に変換 (transform) を適用することもできる. boost::proto では, パターンマッチングによって変換を行うことができる. これらの機能により, メタプログラミングによって最適化を行うようなライブラリの作成において boost::proto が有用であることが期待される.

3. 融合変換機能付きライブラリの実現

3.1 対象とするプログラム

本研究では, 融合変換のうち最も単純な規則である map-map 融合則を例題として用いる. map 計算は, パラメータの関数を入力リストのそれぞれの要素に適用して, 得られた結果をリストとして返す計算である. map 計算は, 関数型言語 Haskell の記法を用いるとインフォーマルには次のように定義される.

$$map\ f\ [a_1, a_2, \ldots, a_n] = [f\ a_1, f\ a_2, \ldots, f\ a_n]$$

map-map 融合則は, 2 つの連続した map 計算を 1 つの map 計算に融合できるというもので, 次の式で表現される. ただし, ∘ は合成関数を作る演算子である.

$$map\ f \circ map\ g = map\ (f \circ g)$$

準備として, 整数の配列をラップした構造体 vect と, map 計算のための関数 map の単純な定義を図 1 に示す. これらの構造体と関数を用いたユーザプログラムの例を図 2 に示す. ユーザプログラムでは, ある数 a による積を表す関数オブジェクト mult を用いて, 配列の各要素に 77 と 99 を掛ける. 図 1 と図 2 を組み合わせたプログラムでは, map 関数の中のループが 2 回実行される.

3.2 式テンプレートを直接記述した融合評価

boost::proto を用いた実装について述べる前に, 式テンプレートを直接記述することで融合評価を行うプログラムを示す. このアイデアは, 著者らが開発している「助っ人」ライブラリでも用いているものである [10].

図 3 に, 式テンプレートを直接記述して融合評価を行うプログラム (の差分) を示す. このプログラムでは, ネストした map 計算の式から, 関数とその適用先をパラメータとする構造体 mapobj を用いて, 計算木を表すオブジェクトを作る. 例えば,

```
map(mult(99), map(mult(77), xs))
```

という式からは,

```
mapobj<mult, mapobj<mult, vect> >
```

という型を持つオブジェクトが生成される. mapobj 構造体は, 配列と同じように [] 演算子をオーバーロードしており, その中で計算木の構造をたどって計算を行う. myeval 関数は, XS 型の計算木を受け取り, 1 回のループによって計算木のすべてを評価する.

GCC (バージョン 4.6.3, 最適化 -O3) において

```
vect ys = myeval(map(mult(99),
                     map(mult(77), xs)));
```

というプログラムより

```
vect ys;
for (int i = 0; i < size; i++) {
  ys[i] = 99 * (77 * xs[i]); }
```

と等価なコードが生成される.

3.3 boost::proto での抽象構文木の構築と評価

以下では, boost::proto を用いて融合変換機能付きライブラリを実現する手法を 2 つ試す. その前に, まず, 融合変換を行わずに, 抽象構文木を構築してそれを単純に評価するプログラムを示す.

図 4 は, boost::proto を用いて抽象構文木を作る部分のプログラムである. boost::proto では, C++の演算子からなる式から抽象構文木を構築するよう, 演算子をオーバーロードして定義している. 一方, ここでの対象となる map 関数のように関数に対してはライブラリ開発者が抽象構文木の構築するプログラムを記述する必要がある. ここでは, 汎用的な関数を表す proto::tag::function を用いて抽象構文木を構築している. 抽象構文木のノードの第 1 子は map 計算であることを表す tag_map, 第 2 子は引数の関数オブジェクト, 第 3 子はその前の計算を表す抽象構文木である. boost::proto における構文木では, すべての要素が, 非終端ノードまたは終端ノードを表す型を持つ必要があるため, テンプレートの特殊化を利用して配列に対し proto::tag::terminal を付加している.

図 5 は, 構築された抽象構文木をもとに融合せずに計算を行うプログラムである. 抽象構文木を評価するための unfused_context という文脈を定義している. その内訳は, 終端ノード (配列) の場合の評価方法 (5–7 行目) と, 非終端ノードの場合の評価方法 (9–17 行目) からなる. ここで, 非終端ノードの場合において, その map 計算の関数オブジェクトを C++の標準的な評価方法である default_context() を用いている. これは, 単に終端ノードに格納されている値を取り出す操作を行う. その結果得られる関数オブジェクトの型は, C++11 の型推論 (auto) によって決まる[*1].

[*1] 明示的に型を計算する式を記述することで, C++11 の型推論を使わずに記述することも可能だと予想するが, 確かめていない.

3.4 boost::proto での融合評価による融合

融合変換機能付きライブラリの実現方法の 1 つ目として, 第 3.2 節で示した融合評価による実現を示す. 構文木の構築については, 図 4 に示したプログラムと同じものを用いる.

異なる点は, 構文木を評価する文脈に別のものを用いる点である. 図 6 に, 融合評価のための文脈 fused_context の定義を示す. この文脈は, 抽象構文木をたどり, index で示された要素の値のみを計算する (7–15 行目). 第 3.2 節で示した融合評価と同様に, myeval 関数において繰り返しを行い, 各要素に対して, fused_context を適用して計算する.

GCC (バージョン 4.6.3, 最適化 -O3) において

```
vect ys = myeval(map(mult(99),
                     map(mult(77), xs)));
```

というプログラムのコンパイル結果は, 第 3.2 節の結果と同じとなった.

3.5 boost::proto での抽象構文木の変換による融合

融合変換機能付きライブラリの 2 つ目の実現として, 抽象構文木を変換することによって融合する. すなわち, *map* $f \circ$ *map* g の形の抽象構文木から *map* $(f \circ g)$ の形の抽象構文木へ変換する.

この抽象構文木の変換による融合変換を実現するプログラムを図 7 に示す. 抽象構文木の構築およびその評価は, 図 4 と図 5 に示したものと同じである.

図 7 の 1–7 行目は, 合成関数に相当する関数オブジェクトの定義である. 9–32 行目の構造体 MapMapFusion が, 抽象構文木の変形を定義したものである. この定義は, パターンマッチと変換の組 2 つからなる. 1 つ目のパターンマッチ (14–15 行目) に該当すると, ネストした map 計算を合成関数による map 計算に変換する (17–26 行目). 2 つ目のパターンマッチ (29 行目) は, その他のすべてにマッチして, 抽象構文木をそのまま返す. 1 つ目のパターンマッチにおいて, 2 つの関数オブジェクト f と g の合成関数は, proto::plus を用いて f + g に対応する抽象構文木によって表現している. boost::proto において, 抽象構文木の変換を行う機構は複雑なテンプレートメタプログラミングによって実現されているため, 新しい構造を返すようなプログラムは現時点ではこのような長いプログラムとなってしまっている.

また, 合成関数を表現する抽象構文木に proto::plus を用いているため, ユーザが使う関数オブジェクトについて, 合成関数のオブジェクトを生成するよう演算子 + をオーバーロードする必要がある (図 8). これについては, 合成関数を特定のタグで表現して, 適切な文脈を記述すること

表 1 各プログラムのライブラリ部分の大きさ

項目	行数	(バイト)
(1). `vect` + `map`	17	(0.4k)
(2). (1) + 式テンプレート直接	37	(0.8k)
(3). (1) + proto 抽象構文木	55	(1.8k)
(4). (3) + 融合評価	79	(2.4k)
(5). (3) + 抽象構文木の変形	78	(2.5k)

で回避できると考える[*2].

図 7 と図 8 に示したプログラムは, GCC (バージョン 4.6.3, 最適化 `-O3`) において第 3.2 節の結果と同じとなった.

4. 議論

4.1 boost::proto を用いることの利点と問題点

boost::proto は, ドメイン特化組込言語のためのテンプレートメタプログラミングを支援するために開発されたライブラリである. 一方, 本研究での目的は特定の計算パターンに対する最適化機能の実現であり, 少しずれた目的となっている. その理由もあってか, 記述しようとするものが boost::proto で提供されている範囲であれば非常に書き易く, 少しでも外れてしまうと複雑もしくは大変困難になってしまうこととなった.

まず, ライブラリに相当する部分のプログラムの大きさについて, 表 1 にまとめる. 行数やバイト数は, コメントと空行, 行頭の空白を除去してカウントしたものである. 式テンプレート `mapobj` を直接記述したものと比べると, boost::proto を用いて作成したプログラムは倍以上の大きさとなった. 一方で, boost::proto を用いた 2 つの実現法の間では, プログラムの大きさにそれほど差がない結果となった.

抽象構文木がすべてテンプレートによる型で表現されることから, boost::proto を用いたライブラリ開発の途中で間違えると, 非常に長いエラーメッセージが出力される. また, 型の計算などのテンプレートメタプログラミングの手法を習熟していないと, エラーの原因を把握・修正することが非常に困難である. プログラムの大きさと開発の困難さの観点から, 現時点では著者らは著者らが開発した「助っ人」ライブラリの融合変換機能 [5], [10] を boost::proto で再実装することはその労力に見合わないと考えている.

ただし, 既存の式テンプレートを直接記述する手法に比べて, boost::proto により抽象構文木を扱うと次の 2 つの利点がある.

- 抽象構文木から複数の値の計算
- 抽象構文木の変換による実装の切り替え

最適化を行うにあたり, 抽象構文木から複数の値を計算する必要があることが多くある. 例えば, 著者らの融合変換機能の実装 [10] では, 融合評価のためのオブジェクトだけでなく, 融合が可能かどうかを判定するために複数の値を計算していた. boost::proto では, 抽象構文木とその上の計算は完全に独立しているため, 文脈を追加すれば簡単にそのような追加の値を計算できる. これを利用することで, 最適化の条件を見通し良く記述できることが期待できる. ライブラリに含まれる計算パターンや最適化規則が増えるときには拡張性の高さが重要となるだろう.

第 3.5 節で示したとおり, boost::proto を用いると抽象構文木を変換することができる. 大域的なプログラムの性質を利用した最適化を行うには, この抽象構文木の変換が役立つと期待できる. 例えば, scan (prefix-sum) 計算を並列に実現するには, [局所 reduce; 大域 scan; 局所 scan] と [局所 scan; 大域 scan; 局所 map] の 2 つの方法が知られている. これら 2 つの実装のどちらが高速かは, その前後の計算に依存する. 抽象構文木に対するパターンマッチを利用すれば, これらのアルゴリズムの切り替えをライブラリに行わせることが可能となる. 並列計算において通信をまたがった構造の変換 [8] や, アルゴリズムの計算量を変えるような強力なプログラム変換 [4] を実現するには, 抽象構文木の変換が必要となるだろう.

4.2 関連研究

プログラムをプログラムするメタプログラミングのための機能や言語は複数提案されている. 例えば, 関数型言語 Haskell では, 簡単な記述によりプログラム変換を実現することができる Rewriting Rules (RULES Pragma)[14] や, Haskell のプログラムを操作する Template Haskell [15] がある. C++言語では, メタプログラミングを行うための言語環境として OpenC++が提案・実装された [2]. 著者らは, この OpenC++を用いて, 最適化機能を持つトランスレータを実現した [11], [18]. OpenC++は, ユニーク番号の生成などコンパイラで有用な機能を持っていたため, map-map 融合よりずっと複雑な抽象構文木の変換であっても比較的容易に実装が可能であった. 近年では, C++におけるテンプレートを用いたメタプログラミング手法が広く利用されている.

最適化機能を持つようなライブラリの開発について, 複数の研究がある. 式テンプレートを用いた線形代数計算ライブラリとして有名なものに uBLAS [17] がある. また, NT2 [6] は, さらにマルチコア環境などの並列計算環境における線形代数計算ライブラリとして実装されているものである. 並列計算における計算パターンを抽象化した並列スケルトンの研究分野では, 最適化機能を持たせたライブラリの研究がこれまで複数行われている. Aldinucci らによる FAN フレームワーク [1] は, 大域的な並列スケルトンの変換ルールを定義し最適化を行うものである. Iwasaki ら [8] は, 並列スケルトンに対して系統的な最適化を行うた

[*2] 合成関数の型はライブラリ開発時には分からないため, その型を計算する必要がある. そのような計算を行うプログラムの作成に挑戦したものの, 現時点ではできていない.

めの規則を与えており, 著者らはそれらの規則を実装した並列スケルトンライブラリの開発を行った [11], [18]. 式テンプレート機能による融合評価を用いた並列スケルトンライブラリの実装には, 著者ら [5], [10] が実装したもの以外に, OSL [9] がある.

5. まとめと今後の課題

本論文では, map-map 融合を例題として, 最適化機能付きライブラリの実現に boost::proto を用いることの利点と問題点について議論した. 既存の式テンプレートを用いた融合評価の最適化機能 [10] の実現手法については, boost::proto を用いることにより見通し良いプログラム作成ができるものの, プログラム記述量と開発途中でのエラー発見の困難さで劣るという知見を得た. また, boost::proto を用いて抽象構文木を変換することで最適化を行うこともできることを示した. ただし, 抽象構文木を自由に変換できるようにするには, まだ解決すべき点が残っている. この問題が解決できたならば, boost::proto を用いることでより拡張性の高い最適化機能付きライブラリ開発が可能となると期待される.

本研究では, map-map 融合という最も単純なプログラム変換を対象とした. 著者らが開発している「助っ人」ライブラリ [5], [10] では, 内部でより複雑な融合変換を行っているため, それらがすべて移植できるかはさらに調査が必要である. また, 抽象構文木の変換における問題点については, boost::proto のドキュメント [13] にも詳しく述べられていないため, 結果もしくは改良案をフィードバックすることも今後の課題となる.

謝辞 本研究の一部は, 科学研究費補助金 12910905, および, 独立行政法人科学技術振興機構「日本 (JST) — フランス (ANR) 研究交流」 10102704 (JST) / ANR-2010-INTB-0205-02 (ANR) により行われた.

参考文献

[1] Aldinucci, M., Gorlatch, S., Lengauer, C. and Pelagatti, S.: Towards parallel programming by transformation: the FAN skeleton framework, *Parallel Algorithms and Applications*, Vol. 16, No. 2–3, pp. 87–121 (2001).

[2] Chiba, S.: A Metaobject Protocol for C++, *OOPSLA'95, Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 285–299 (1995).

[3] de Guzman, J., Marsden, D. and Heller, T.: Phoenix 3.0, In Chapter 1 of boost Library Documentation, available from http://www.boost.org/doc/libs/1_53_0/libs/phoenix/doc/html/index.html (2010).

[4] Emoto, K., Fischer, S. and Hu, Z.: Generate, Test, and Aggregate - A Calculation-based Framework for Systematic Parallel Programming with MapReduce, *Proceedings of 21st European Symposium on Programming, ESOP 2012*, Lecture Notes in Computer Science, Vol. 7211, pp. 254–273 (2012).

[5] Emoto, K. and Matsuzaki, K.: An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo, Submitted to International Symposium on High-level Parallel Programming and Applications (HLPP 2013) (2013).

[6] Falcou, J., Sérot, J., Pech, L. and Lapresté, J.-T.: Metaprogramming Applied to Automatic SMP Parallelization of Linear Algebra Code, *Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008, Proceedings*, Lecture Notes in Computer Science, Vol. 5168, pp. 729–738 (2008).

[7] Gill, A. J., Launchbury, J. and Jones, S. L. P.: A Short Cut to Deforestation, *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pp. 223–232 (1993).

[8] Iwasaki, H. and Hu, Z.: A New Parallel Skeleton for General Accumulative Computations, *International Journal of Parallel Programming*, Vol. 32, No. 5, pp. 389–414 (2004).

[9] Javed, N. and Loulergue, F.: OSL: Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays, *8th international Conference on Advanced Parallel Processing Technologies (APPT'09)*, Lecture Notes in Computer Science, Vol. 5737, pp. 436–451 (2009).

[10] Matsuzaki, K. and Emoto, K.: Implementing Fusion-Equipped Parallel Skeletons by Expression Templates, *Implementation and Application of Functional Languages, 21st International Workshop, IFL 2009, Revised Selected Papers*, Lecture Notes in Computer Science, Vol. 6041, pp. 72–89 (2010).

[11] Matsuzaki, K., Kakehi, K., Iwasaki, H., Hu, Z. and Akashi, Y.: A Fusion-Embedded Skeleton Library, *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Proceedings*, Lecture Notes in Computer Science, Vol. 3149, pp. 644–653 (2004).

[12] Niebler, E.: Boost.Xpressive, In Chapter 37 of boost Library Documentation, available from http://www.boost.org/doc/libs/1_53_0/doc/html/xpressive.html (2007).

[13] Niebler, E.: Boost.Proto, In Chapter 22 of boost Library Documentation, available from http://www.boost.org/doc/libs/1_53_0/doc/html/proto.html (2008).

[14] Peyton Jones, S., Tolmach, A. and Hoare, T.: Playing by the Rules: Rewriting as a practical optimisation technique in GHC, *Proceedings of Haskell Workshop 2001* (2001).

[15] Sheard, T. and Peyton Jones, S.: Template metaprogramming for Haskell, *Proceedings of Haskell Workshop 2002*, pp. 1–16 (2002).

[16] Wadler, P.: Deforestation: transforming programs to eliminate trees, *Theoretical Computer Science*, Vol. 73, No. 2, pp. 231–248 (1990).

[17] Walter, J., Koch, M., Winkler, G. and Bellot, D.: Basic Linear Algebra Library, In boost Library Documentation, available from http://www.boost.org/doc/libs/1_53_0/libs/numeric/ublas/doc/index.html (2010).

[18] 明石良樹, 松崎公紀, 岩崎英哉, 筧 一彦, 胡 振江: 最適化機構を持つ C++並列スケルトンライブラリ, コンピュータソフトウェア, Vol. 22, No. 3, pp. 214–222 (2005).

```
1  struct vect {
2    int        data[SIZE];
3    int&       operator[](int i)       { return data[i]; }
4    const int& operator[](int i) const { return data[i]; }
5    void print() const { ... }
6  };
7
8  template <typename F>
9  vect
10 map(const F f, const vect &xs) {
11   vect ret;
12   for (int i = 0; i < size; i++) ret[i] = f(xs[i]);
13   return ret;
14 }
```

図 1 配列のためのデータ構造 vect と関数 map の単純な実装

```
1  struct mult {
2    typedef int result_type;
3    int a;
4    mult(int a_) : a(a_) {};
5
6    int operator()(int x) const { return a * x; }
7  };
8
9  int main() {
10   vect xs; xs[0] = 2; xs[1] = 3;
11
12   vect ys = map(mult(99), map(mult(77), xs));
13
14   ys.print(); return 0;
15 }
```

図 2 ある数 a による積を表す関数オブジェクト mult と, 関数 map を用いたプログラム

```
template<typename F, typename XS>
struct mapobj {
  const F f;  const XS &xs;
  mapobj(const F& f_, const XS& xs_) : f(f_), xs(xs_) {}
  int operator[](int i) const { return f(xs[i]); }
};

template<typename F, typename XS>
mapobj<F, XS>
map(const F f, const XS& xs) {
  return mapobj<F, XS>(f, xs);
}

template<typename XS>
vect
myeval(const XS& xs) {
  vect ret;
  for (int i = 0; i < size; i++) ret[i] = xs[i];
  return ret;
}
```

図 3 式テンプレートを直接記述した融合評価

```
struct tag_map {};

template<typename F, typename Arg>
typename proto::result_of::make_expr<proto::tag::function,
           typename proto::result_of::make_expr<proto::tag::terminal, tag_map>::type,
           typename proto::result_of::make_expr<proto::tag::terminal, F>::type,
           Arg
         >::type const
map(const F f, const Arg &arg) {
  return proto::make_expr<proto::tag::function>(
                  proto::make_expr<proto::tag::terminal>(tag_map()),
                  proto::make_expr<proto::tag::terminal>(f),
                  arg);
}

template<typename F>
typename proto::result_of::make_expr<proto::tag::function,
           typename proto::result_of::make_expr<proto::tag::terminal, tag_map>::type,
           typename proto::result_of::make_expr<proto::tag::terminal, F>::type,
           typename proto::result_of::make_expr<proto::tag::terminal, const vect&>::type
         >::type const
map(const F f, const vect &xs) {
  return map(f, proto::make_expr<proto::tag::terminal>(boost::ref(xs)));
}
```

図 4 boost::proto による抽象構文木の構築

```
struct unfused_context
                : proto::callable_context<const unfused_context, proto::null_context> {
  typedef vect result_type;

  vect operator()(proto::tag::terminal, vect xs) const {
    return xs;
  }

  template<typename F, typename XS>
  vect operator()(proto::tag::function, typename proto::terminal<tag_map>::type const&,
                  F f, XS xs) const {
    vect ys = proto::eval(xs, *this);
    auto func = proto::eval(f, proto::default_context());

    vect ret; for (int i = 0; i < size; i++) ret[i] = func(ys[i]);
    return ret;
  }
};

// User program
int main()
{
  ...

  auto prog = map(mult(99), map(mult(77), xs));
  vect ys = proto::eval(prog, unfused_context());
```

図 5 融合なしでの抽象構文木の評価

```
struct fused_context
              : proto::callable_context<const fused_context, proto::null_context> {
  typedef int result_type;
  const int index;
  fused_context(int index_) : index(index_) {};

  int operator()(proto::tag::terminal, const vect &xs) const {
    return xs[index];
  }

  template<typename F, typename XS>
  int operator()(proto::tag::function, typename proto::terminal<tag_map>::type const &,
                 F f, XS xs) const {
    return proto::value(f)(proto::eval(xs, *this));
  }
};

template<typename Expr>
vect myeval(Expr &expr) {
  vect ret;
  for (int i = 0; i < size; i++) ret[i] = proto::eval(expr, fused_context(i));
  return ret;
}
```

図 6 抽象構文木の融合評価

```
template <typename F, typename G>
struct comp {
  const F f; const G g;
  comp(const F& f_, const G& g_) : f(f_), g(g_) { }
  typedef int result_type;
  int operator()(int x) const { return f(g(x)); }
};

struct MapMapFusion
  : proto::or_<
  // case 1: map(f, map(g, xs))     --- f, g, xs are parameters
  proto::when<
    // if the following pattern matches,
    proto::function<proto::terminal<tag_map>, _,
                    proto::function<proto::terminal<tag_map>, _, _> >,
    // transform it into an object of type
    proto::function< proto::_left,
             proto::plus<proto::_child_c<1>, proto::_child_c<1>(proto::_child_c<2>) >,
             proto::_child_c<2>(proto::_child_c<2>) >
    (
     // with the following values
     proto::_left,
     proto::plus<proto::_child_c<1>, proto::_child_c<1>(proto::_child_c<2>) >(
                          proto::_child_c<1>, proto::_child_c<1>(proto::_child_c<2>)),
     proto::_child_c<2>(proto::_child_c<2>)
    )>
  ,
  // case 2: otherwise
  proto::when<_,
              proto::_expr>
  >
{};
```

図 7 抽象構文木の変形による融合変換

```
struct mult {
  ...

  template<typename G>
  comp<mult, G>
  operator+(const G& g) const { return comp<mult, G>(*this, g); }
};

int main()
{
  ...

  auto prog = map(mult(99), map(mult(77), xs));
  vect ys = proto::eval(MapMapFusion()(prog), unfused_context());
```

図 8 抽象構文木の変形の実行