C++ テンプレートを分割コンパイルするためのアプローチ

増山 隆[†] 住井 英二郎[†] 米澤 明憲[†]

既存の C++ テンプレートのコンパイルモデルは，インライン化によるものである．このモデルでは，各オブジェクトコード，実行可能コードのサイズが大きくなりがちであり，またテンプレートで書かれた実装が隠蔽できないという問題がある．本論文では，C++ のテンプレートを分割コンパイルする方法を示す．この方法は C++ のソースコードからソースコードへの変換に基づいている．変換後のコードは実行時に，(1) テンプレートの parametric 多相性を扱うために，型多相なデータに対し，(多くの ML コンパイラが行うように) boxing と unboxing の操作をする．(2) C++ の ad hoc 多相性を扱うために (Haskell コンパイラが行うように) オーバーロードされた関数を渡す．また，型多相なオブジェクトに対するメソッド起動を扱うために，C++ の多重継承と抽象インターフェイスの機能を用いる．

An Approach to Separate Compilation of C++ Template

Takashi Masuyama, Eijiro Sumii and Akinori Yonezawa

The existing compilation model of C++ templates—namely inlining —suffers from a problem that each object code and the resulting executable tend to become huge, and also, that implementations of templates cannot be hidden. In this presentation, we propose a method of separately compiling C++ templates. This method is based on source-to-source transformation of C++ code. At runtime, the transformed code (1) performs boxing and unboxing on polymorphic data (as many ML compilers do) in order to deal with the parametric polymorphism of templates and (2) passes around overloaded functions (as Haskell compilers do) in order to treat the ad hoc polymorphism of C++. Our method also exploits the multiple inheritance and abstract interface mechanisms of C++ to handle method invocations on polymorphic objects.

1. 序 論

1.1 パラメトリック多相性の意義

プログラマはプログラムを型多相に書くことによって，データ型に依らずにアルゴリズム自身を記述できる．一度アルゴリズムが多相型を用いたスタイルで書くと，異なる型のデータに対して適用することができる．よって，多相型プログラムはソースの記述量を減らし，開発を効率良くする．C++ のテンプレートはパラメトリック多相性をもつプログラミングを可能にする．Standard Template Library[1)] はテンプレートを用いて型多相なリスト，集合などの基本的で有用なデータ構造を提供している．

1.2 分割コンパイルの意義

一方，分割コンパイルはコンパイル済みのコードを再コンパイルすること無しに，再利用することを可能にする．更に分割コンパイルによって，一度にコンパイルされるファイルサイズは小さくなるので，コンパイラが必要とする CPU 時間やメモリ領域は小さくてすむ．最後のコンパイルから，編集されたファイルと，そのファイルに依存するファイルのみ再コンパイルされればよい．大規模プログラミングではこの性質は重要である．

1.3 パラメトリック多相性と分割コンパイルの両立の困難さ

しかし，一般にパラメトリック多相性と分割コンパイルは両立が困難である．型多相な定義をコンパイルするときには，型多相なオブジェクトの具体的な型が分からないからである．

オブジェクトの表現もサイズも型によって異なる．また，型によって異なった呼び出し規則が使われる．よって，型多相な関数をコンパイルする時点では，型多相なオブジェクトを何バイト，どの位置に割り当てるのがよいかは分からない．

さらに C++ では，多重定義を許している．関数名と引数の具体的な型によって呼び出される関数が特定できるが，テンプレート関数のコンパイル時には，多相型を持つオブジェクトに適用される関数は特定できない．

† 東京大学情報理工学系研究所コンピュータ科学専攻
Department of Computer Science, University of Tokyo, Graduate School of Infomation Science and Technology

実際，ほとんどの C++ コンパイラは多相性を持つ定義をその使用と分割してコンパイルする機能はサポートしていない．それらは，テンプレート関数を使用している箇所をコンパイルする時に，具体的な型で特別化した定義を生成することで解決している．そのため，コンパイルされたコードは巨大になり，コンパイルには多くの時間がかかることが問題となっている．

1.3.1 提案する方式

本論文では，既に確立された ML， Haskell の手法を用いて C++ のテンプレートを分割コンパイルする方法を提案する．

データの表現に関する問題に対しては，多くの ML コンパイラが行うように，ボックス化を行い，統一された表現にして渡す． boxing， unboxing の詳細については 2節で述べる．

また，型多相なオブジェクトに適用される多重定義された関数ついては， Haskell コンパイラが行うようにテンプレート関数の追加の引数として渡して解決する．多重定義された関数の取り扱いや，型多相な構造体，メソッド起動，また多重定義された演算子の取り扱いに関する工夫については 3節で述べる．

多重定義された関数を追加の引数として受け取ることで，従来のテンプレートの宣言には現われなかった “多相型 T に定義されているべき関数” が変換後の関数の宣言に必要となる． テンプレートのインターフェイスについての詳細は 5節で述べる．

本論文では C++ のテンプレートを分割コンパイル可能にし，オブジェクトコード，実行可能コードのサイズを小さくすることを目的とする．実際に本論文で変換し，コンパイルしたコードと，従来の方法でコンパイルしたコードの実行効率，コードサイズの比較は 6節で行う．

1.4 貢　献

この論文は C++ のテンプレートを分割コンパイル可能にするアプローチを提案する．また、そのアプローチをいくつかのプログラムに対して適用して実験を行った．結果として，従来の方法によってコンパイルされたコードより小さなオブジェクトコードと実行可能コードが得られる．多相性を含むコードは一度コンパイルされればよい．

本論文で述べられている変換器はまだ実装されていない．さらに，ケースバイケースで考えられているので，C++ の仕様の全てには合わないかもしれない． C++ の仕様を網羅することは目的としていない．

2. boxing / unboxing の操作

この論文では，変換前から一語のデータや単相型を持つオブジェクトは直接渡し，多相型を持つオブジェクトは型によらない統一した表現で渡す．ある具体的な型の分かっているオブジェクトを一語の表現に変換する操作が boxing であり，逆に一語の表現をある特定の型の表現に戻す操作が unboxing である．以降では boxing と unboxing が必要性である理由や必要となる箇所について述べる．

2.1 boxing / unboxing が必要となる理由

プログラム中では様々なデータ型が使われる． そして，オブジェクトのサイズはデータ型によって異なる．さらに，呼び出し規約によっては，関数の引数を割り当てる場所が異なるかもしれない．例えば浮動小数点レジスタのあるマシンでは， 浮動小数点数は浮動小数点数レジスタを介して直接受け渡され， 一語で収まるデータは汎用レジスタを介して受け渡され，それ以外はスタックを介するかもしれない。呼び出し側と呼び出される側は型が分かっているから，型に応じた効率のよい呼び出し規約 を使うことができる．

しかし，多相型をもつ関数をその使用と分割してコンパイルしようとすると，多相型をもつ関数をコンパイルする時点では，それがどの型で使用されるかは，分からない．よって， 具体的な型に特化した呼び出し規約や，オブジェクトのサイズの情報は利用できない．

例えば，次のような多相関数 “second_arg” を考えてみる。

```
template <class S, class T>
T second_arg(S x,T y) { return y; }
```

ここでは，メモリ上のスタックを介して引数を受け渡しする呼び出し規約を考える．もし， x が多語のデータとして，スタックにコピーされて積まれたら， y のスタックフレーム内での位置は x のサイズに依存するが，そのサイズは “second_arg” を分割コンパイルする時点では利用できないので， y の位置はコンパイル時には決まらない． x の型によらずに y をスタックから取り出す何らかの方法が必要である．また，この関数は，多相型を持つ返り値 y を返す．このとき，返り値を規約に従った場所にコピーして返さなければならないが， y のサイズもやはり “second_arg” のコンパイル時には分からない．y をコピーする何らかの方法が必要である．

そこで，多相な型を持つ x， y を型によらずに統一して一語で表すことにすると， y のスタックフレーム内での位置はコンパイル時に決まる．さらに，一語をコピーすれば良いので，その一語の表現を返すことで，多相な

データを返すことができる．この一語の表現にはメモリに割り当てられたデータへのポインタを使えばよい．

2.2 boxing / unboxing が必要となる箇所

単相型か多相型かという観点から，引数の受渡しや代入を分類すると次の 4 通りに分けられる．

(1) 単相型のデータを単相型の変数に代入
(2) 単相型のデータを多相型の変数に代入
(3) 多相型のデータを単相型の変数に代入
(4) 多相型のデータを多相型の変数に代入

1の場合は，多相型は関わらないので，我々の変換によって変化しない．4の場合は，テンプレート関数の中で型多相なオブジェクトにテンプレート関数を適用している場合で，この場合は boxing されたまま受渡しすればよい．boxing は 2の場合に必要である．より具体的には以下の状況で必要となる．

- テンプレート関数を単相型のオブジェクトに適用する．(a)
- 高階関数や，多重定義された関数で，具体的な型の分かっているオブジェクトを型多相なオブジェクトとして関数が返す．(b)

同様に unboxing は 3の場合に必要であり，以下の状況で行われる．

- テンプレートから呼ばれる関数の引数は boxing された状態で渡す．それを，具体的な型に対して定義された関数（unboxing されたオブジェクトを扱う）に渡す前．(b)
- 関数の型多相な返り値を具体的な型として使用する．(a)

(a) はテンプレート関数を呼ぶ前後，(b) はテンプレート関数内で型多相なオブジェクトに適用される関数内部での boxnig / unboxing が必要な状況である．以降で例を挙げながら (a),(b) それぞれの場合について説明する．

2.2.1 テンプレート関数の引数として渡す前，返り値を受け取った後の boxing / unboxing

テンプレート関数は型多相なオブジェクトについて型の情報を知らないまま一語の表現で渡されるものとして変換される．この型多相な引数は型によらないように一語の表現に統一する．そして，この関数の呼び出し側がデータを boxing して一語の表現にして渡す．具体的には，多語のデータはメモリに割り当てて，それへのポインタ（一語）をそのデータの表現として使い，もともと一語であるデータはデータそのものを一語のデータとする．例として 2.1節で示した “second_arg” 関数は以下のように変換される．

```
//# template <class S, class T>
//# T second_arg(S,T)
wrapped_t second_arg(wrapped_t x,
                     wrapped_t y)
  { return y; }
```

“wrapped_t” は一語の表現で C++ での型は void* である．テンプレート関数の型引数は “wrapped_t” で置き換える．2 つの引数の型が異なっていてよいか，また同じでなければならないかというテンプレート関数の型引数の情報は，この置き換えによって失われるので，最初のコメントのように表しておく．このコメントは変換器がこの関数への呼び出しを変換するときに用いる．

“second_arg” の適用側は引数を渡す前に boxing しなければならない．次の適用例について考えてみる．

```
void foo() {
  double d = second_arg(1,1.34);
}
```

“foo” のなかで，“second_arg” は int 型と double 型のデータに適用されている．int 型のサイズは一語なので，変換後もそのまま渡す．一方，double 型のサイズは 2 語なので，データをメモリに割り当てて，それへのポインタを渡す．変換後のコードは以下のようになる．以下のソースコードでは式 exp を評価した型 type の値を boxing するプリミティブを “WRAP(type,exp)” とし，boxing された変数 var を型 type で unboxing するプリミティブを “UNWRAP(type,var)” とする．

```
void foo() {
   //copy data itself
   wrapped_t tmp0 =
     (wrapped_t)1;
   //boxing
   wrapped_t tmp1 =
     WRAP(double,1.34);
   wrapped_t tmp2 =
     second_arg(tmp0,tmp1);
  //unboxing
   double d = UNWRAP(double,tmp2);
}
```

この例では “second_arg” は型多相な値を返す．その具体的な型は，テンプレートの型情報から第 2 引数の型（ここでは double）であることは “foo” のコンパイル時には分かっている．double はメモリに割り当てて返されるので，メモリから値を取り出して使う．

2.2.2 多相型を持つ関数に適用される関数の内部での boxing / unboxing

型多相な引数に適用される型多相な高階関数は boxing されたオブジェクトを扱わなければならない．次の例に

ついて考えてみる．

```
template <class S,class T>
T apply(T (*f)(S), S x) {
  return f(x);
}
```

この例では関数 “f” は型多相な引数 x に適用され，型多相な値を返す．このテンプレート関数が変換前に以下のように使われていたとする．

```
double plus_one(double x)
  { return x + 1.0; }
void goo() {
  double d =
    apply(plus_one,2.0);
}
```

関数 “plus_one” は引数を double に特化した規約で受け取るだろう．しかし，“apply” のほうからはその規約は分からない．この差を埋めるために我々の変換方式では boxing されたオブジェクトを引数として受け取り，それをボックス取り出し化して，元の関数 “plus_one” に渡す関数を新たに定義する．さらに，その関数は “plus_one” からの返り値を boxing する．以下の “wrapped_plus_one” がその関数にあたる．この関数は具体的な型の分かっている値（double 型）を boxing して型多相な値として返す．

```
//# template <class S,class T>
//# T apply(T (*f)(S), S x)
wrapped_t
  apply(wrapped_t (*f)(wrapped_t),
                  wrapped_t x) {
  return f(x);
}

wrapped_t wrapped_plus_one(wrapped_t x) {
  double tmp0 =
    UNWRAP(double,x); //unboxing
  double tmp1 =
    plus_one(x);
  double tmp2 =
    WRAP(double,x);   //boxing
  return tmp2;
}

void goo() {
  wrapped_t tmp0 =
    WRAP(double,2.0);
  wrapped_t tmp1 =
    apply(wrapped_plus_one,tmp0);
  double d =
    UNWRAP(double,tmp1);
}
```

この “wrapped_plus_one” のような関数は引数から受け取った関数の適用のみならず，多重定義された関数の呼び出しに対しても，型多相なオブジェクトに対して適用されていれば導入される．多重定義された関数の扱いについては，3節で述べている．

2.2.3　構造体の多相型を持つメンバ

型多相な構造体やクラス（例えば要素が多相型であるペア，リスト，ツリーなど）で，型多相なフィールドは構造体の形が型によらずに同じになるようにするために，boxing されていなければならない．例として図 2 のように定義された “MyPair” とそれを扱う関数 “map_pair” を考える．

変換後のコードは図 3のようになる．ペアの要素 “first”，“second” は boxing される．

“MyPair” の各要素が boxing されているために，“map_pair” は要素の型を気にするすることなく統一された方法で要素を取り出すことができる．

2.3　型多相なオブジェクトを割り当てるメモリロケーション

型多相なオブジェクトは，メモリ上に割り当てる．素朴な方法として，全ての型多相なオブジェクトをヒープに割り当てる方法が挙げられる．ヒープに割り当てる場合，“WRAP”，“UNWRAP” は以下のようなマクロで実現できる．

```
#define WRAP(type,exp) \
  (wrapped_t) new type(exp)
#define UNWRAP(type,var) *(type*)var
```

関数内部で boxing された値は関数の返り値として返らなければスタック上に割り当てることができるが，返り値からポインタをたどって到達できるならばヒープに割り当てなければならない．

2.2.1を次のように使う場合は，boxing する値は “f” の局所変数としてスタックに割り当ててよい．

```
double f() {
  //box x and y
  int x = 1;
  double y = 3.14;
  wrapped_t result = second_arg(x,&y);
  return *result;
}
```

一方で，例えば，2.2.2で変換後に定義した “wrapped_plus_one” は double の返り値は多相型と

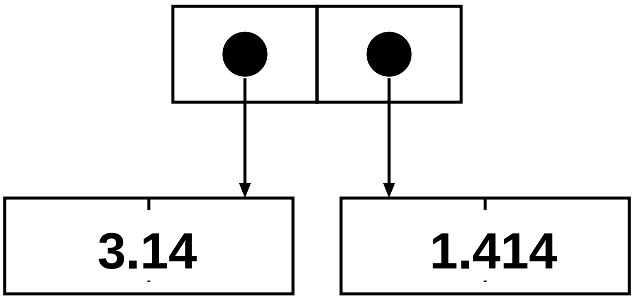

図 1　変換後のペアの表現

```
template <class _T1, class _T2>
struct MyPair {
  _T1 first;
  _T2 second;
  MyPair(const _T1& __a, const _T2& __b) : first(__a), second(__b) {}
};

template <class T1,class T2>
MyPair<T2,T2> map_pair(T2 (*f)(T1),MyPair<T1,T1> data) {
  return MyPair<T2,T2>::MyPair(f(data.first),f(data.second));
}
```

図 2　変換前のペアのソースコード

```
//# template <class _T1, class _T2>
struct MyPair {
  typedef wrapped_t _T1;
  typedef wrapped_t _T2;

  typedef _T1 first_type;
  typedef _T2 second_type;

  _T1 first;
  _T2 second;
  MyPair(const _T1& __a, const _T2& __b) : first(__a), second(__b) {}
};

//# template <class T1,class T2>
//# MyPair<T2,T2> map_pair(T2 (*f)(T1),MyPair<T1,T1> data)
MyPair map_pair(wrapped_t (*f)(wrapped_t),MyPair p) {
  return MyPair(f(p.first),f(p.second));
}
```

図 3　変換後のペアのソースコード

して返すので，値をメモリに割り当てなければならない．この値は “wrapped_plus_one” より寿命が長いので，boxing する値はヒープに割り当てなければならない．この値の占めるメモリ領域を解放するにはガーベージコレクタが必要である．

3. 多重定義とメソッド起動

3.1 多重定義された関数

C++ は関数の多重定義を許しているので，多相型を持つオブジェクトに適用された関数はテンプレート関数のコンパイル時には一意に決定できない．以降で，テンプレート供給側，使用側間での多重定義された関数の受渡しの方法について説明する．

3.1.1 テンプレート供給側

テンプレート関数内で，型多相なオブジェクトに適用されている関数は，テンプレート関数をコンパイルする時点では特定できない．例として，次のテンプレート関数 “foo” について考えてみる．

```
template <class T>
void foo(T x) {
  goo(x);
}
```

関数 “goo” は，型多相なオブジェクト x に対して適用されているので，“foo” のコンパイル時には一意に決定できない．

そこで，上の “goo” のように一意に特定できない関数を，テンプレート関数の引数として受け取るように変換する．この “foo” 関数を呼び出す側からは，この関数 “goo” を新たな引数として渡さなければならないことと，“goo” 関数の引数の型が分かっている必要があるので，その情報を変換後のヘッダファイルに含めておく．

```
//# template <class T> void foo(T x)
//# require void goo(T)
void foo(T x,void (*goo)(wrapped_t)) {
  goo(x);
}
```

上の例で “//#” で始まる行で書かれた情報は実際はインターフェイスとしてヘッダファイルに書いておく．

3.1.2 テンプレート使用側

テンプレート関数の型多相な引数に具体的な型の分かったオブジェクトを与えて呼び出す場合，テンプレート関数のコンパイル時に特定できなかった関数を特定できるようになる．適用側で，テンプレート関数が多相型に対して要求している関数を，具体的型情報から特定して渡す．

次の例について考えてみる．

```
template <class T>
T max(T x, T y)
  { return x > y ? x : y; }

template <class S>
S array_max(S array[],int size) {
 S result = array[0];
 for(int i = 1; i < size; i++) {
   result = max(array[i],result);
 }
  return result;
}


void zoo(int array[], int size) {
  int max_value =
     array_max(array,size);
}
```

この例では，テンプレート関数 “array_max” がテンプレート関数 “max” を呼んでいる．さらに，テンプレート関数 “max” も，その引数の型に対して定義された演算子 “>” を使用している．関数 “array_max” は，“max” を多相型 S をもつ配列要素を引数に与えて呼び出している．この “max” 関数を呼び出し側から受け取るために，“array_max” には，新たな引数として型 S に対して定義された “max” を付け加える．“max” に演算子 “>” を追加の引数として加えるのも同様である．3.1.1節で述べたようにテンプレート関数 “max”，“array_max” は図 4のように変換される．☆

テンプレート関数を使用している関数 “zoo” について考えてみる．関数 “zoo” は “array_max” を単相型（int の配列）で呼び出している．この時点で，型変数に代入される具体的な型が分かるので，テンプレート関数が要求する関数を生成し渡す．まず，“array_max” は関数 “int max(int,int)” を要求している．変換器は “max” 関数の実際の定義を探す．この例では，テンプレート関数として，定義されている “max” が呼ばれるとする．次にテンプレート関数 “max” 変換後の関数を見ると，“max” は int 型の引数を 2 つ取る演算子 “>” を必要としていることが分かる．多相型を持つオブジェクトは boxing されて “greater_than” に渡されるので，これを unboxing して，int の変数に入れて，実際に int に対して定義された演算子 “>” に渡す関数を図 5のように定義する．

☆ 関数 “max” で用いる演算子 “>” は “greater_than” に名前を付け変えている．

```
//# template <class T> max(T x,T y)
//# require bool operator>(T,T) : greater_than
wrapped_t max(wrapped_t x, wrapped_t y,
              bool (*greater_than)(wrapped_t,wrapped_t)) {
  return greater_than(x,y) ? x : y;
}

//# template <class S>
//# require S max(S,S)
//# require S operator[](S*,int) : array_ref : right-value
wrapped_t array_max(wrapped_t array,int size,
                    wrapped_t (*max)(wrapped_t,wrapped_t),
                    wrapped_t (*array_ref)(wrapped_t,int)) {
  wrapped_t result = array_ref(array,0);
  for(int i = 1; i < size; i++) {
    result = max(array_ref(array,i),result);
  }
  return result;
}
```

図 4　“max”, “max_array” の変換後のソースコード

```
bool operator_greater_than_on_wrapped_int(wrapped_t wa,
                                          wrapped_t wb) {
  int a = (int)wa;
  int b = (int)wb; //1 word data
  return a > b;
}
```

図 5　boxing されたオブジェクトを特定の関数に unboxing して渡す関数

続いて “array_max” の変換後の関数の要求を見ると，配列の要素型に対する 2 引数の “max” 関数を要求していることが分かる．“array_max” からは，“max” がテンプレートで定義されているか否かは，“array_max” は知らない．よって図 6のような “max” 関数の要求を満たした関数を定義した 2 引数の関数を新たに定義し，それを “array_max” に渡す．配列アクセスの演算子 “[ ]” も “array_ref” という関数で与える．配列参照，代入についての工夫については 3.3.1節で述べている．

そして，テンプレート関数 “array_max” を使用する関数 “zoo” は図 7のように変換される．

3.2　メソッド起動とフィールドアクセス

もし，多相なオブジェクトに対してメソッドが呼ばれていたら，この多相なオブジェクトはそのメソッドと同名のメソッドをもつあるクラスのインスタンスである．例として次のソースコードについて考えてみる．

```
template <class T>
void print_something(T* printer)
{
  printer->print();
}
```

この例では，“printer” は “print” という名前のメソッドを持つクラスのインスタンスでなければならない．しかも “print_something” からアクセスできるメソッドである．逆に “print” という名の “print_something” からアクセス可能なメソッドをもつクラスのインスタンスであれば，何でもこの関数に適用することができる．そこで，型変数を以下の “T_with_print” のように “print” という名のメソッドをもつクラスに置き換える．

```
class T_with_print {
  protected:
    T_with_print() {}
    ~T_with_print() {}
```

```
wrapped_t binded_max(wrapped_t x, wrapped_t y) {
  //fill a requirement
  max(x,y,operator_greater_than_on_wrapped_int);
}
```

図 6 テンプレート “max” の要求を満たした関数

```
wrapped_t array_ref_on_int(wrapped_t array, int i) {
  int* tmp0 = (int*)array;
  return (wrapped_t)tmp0[i];
}

void zoo(int array[],int size) {
  wrapped_t tmp0 = (wrapped_t)array; //whole array is boxed
  wrapped_t tmp1 = array_max(tmp0,size,
                             array_ref_on_int,binded_max);
}
```

図 7 テンプレート関数を使用する関数 “zoo” の変換後のコード

```
  public:
    virtual void print() = 0;
};
```

メソッド “print” は純粋仮想関数にする．つまりこのクラスを継承するクラスによって，オーバーライドされ，実装される． “T_with_print” クラス自身は，直接のインスタンスを持たない．そして，テンプレート関数 “print_something” 自体を以下のように変換する．

```
//# template <class T>
//# require void T::print()
void print_something(T_with_print*
                               printer) {
  printer->print();
}
```

“print_something” の呼び出しを見付けたら，変換器は “T_with_print” を継承し，実装が変換前の “print_something” の引数の型（あるクラス）のものと同じクラスを定義する．例として次のような呼び出し側のコードを考える．

```
class Printer {
  public:
    Printer(int x) { //... }
    void print();
};

//implementation
void Printer::print() { //... }
```

```
void foo(Printer* p)
{
  print_something(p);
}
```

このコードは以下のように変換される．

```
class T_with_print_and_Printer
  : public T_with_print,
    public Printer {
  public:
    T_with_print_and_Printer(Printer* p)
      : m_printer(p) {}
    ~T_with_print_and_Printer() {}
    void print();
    Printer* m_printer;
};

void T_with_print_and_Printer::print()
{
  //call original method
  // on original object
  m_printer->print();
}
void foo(Printer *p)
{
  print_something
    (new T_with_print_and_Printer(p));
}
```

関数 “foo” の引数の型は変換によって新たに定義

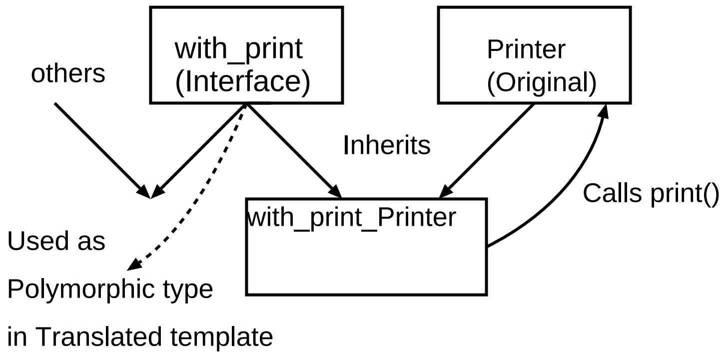

図 8　変換前の引数のクラスと変換によって定義されたクラスとの関係

された “T_with_print_and_Printer” に変えたくはない．もし “T_with_print_and_Printer” に変更してしまうと，“foo” の適用を書き換えなければならないからである．“foo” の引数の型を変えないために “T_with_print_and_Printer” はコンストラクタで Print のインスタンスへのポインタを受け取り，それをフィールドとして保持する．“T_with_print_and_Printer” のメソッド “print” は，そのインスタンスの “print” メソッドを呼び出す．フィールドアクセスはメソッド起動とほぼ同じ方法で扱うことができる．しかし，ある名前のフィールドをもつクラスを抽象的に表すことができないので，その変数へのアクセッサをインターフェイスとなるクラスで定義しておいて，実際それを継承するクラスで，アクセッサの実装を与えることにする．

3.3　演算子の多重定義

C++ は関数のみならず一部の演算子をも多重定義することを許している．演算子 “=”，“[ ]” も多重定義されうる． もしこれらの演算子を関数と同様に扱うと多相性を持つコードの実行効率は著しく落ちる．これらの演算子が多重定義されていないと仮定すると，効率の低下を抑える工夫が考えられる．

3.3.1　演算子 “[ ]”

演算子 “[ ]” は配列に対して適用されるが，例えば STL (Standard Template Library) の vector，set，map のように連想コンテナに対して使われることもある．この演算子はユーザーが多重定義することができる．つまり，演算子 “[ ]” が使われているからといって，それは配列であるとは限らない．

もしこの演算子を，連想コンテナとして扱うとするならば，代入，参照を行う間数を用意して，多重定義された関数と同様に扱うことができる．例えば配列をこの方法で扱う場合 “something[i] = 1;” という左辺値を返す演算子 “[ ]” を使った文は関数呼び出し “set(something,i,1);” に変換される．ここで，関数 “set(array,i,value);” は配列 array の i 番目に value を代入する関数であるとする．そして，右辺値を返す演算子 “[ ]” を使った文 “foo(something[i]);” は “foo(get(something,i))” のように変換される．ここで関数 “get(array,i)” は配列 array の i 番目の要素を値として返す関数であるとする．このアプローチは，演算子 “[ ]” が多重定義された型のオブジェクトと配列を同じコードで扱うことを可能とする．しかし，関数呼び出しのオーバーヘッドが実行効率を落とす．配列に頻繁にアクセスするプログラムでは特に大きくなる．

演算子 “[ ]” が使われているものを長さが分かっている配列であると仮定するとテンプレート関数の型多相な引数として渡す前に配列全体を boxing したオブジェクトの配列に変換して渡すことが可能である．このようにすると，テンプレート関数内では要素サイズが一語の配列を扱えばよい．このアプローチは配列要素の整列のような配列アクセスが多いプログラムに対して有効である．しかし，このアプローチでは関数の呼び出し前と呼び出し後で（boxing されたオブジェクトの配列を返すならば）配列の長さに比例する boxing，unboxing のオーバーヘッドがかかる．配列にほとんどアクセスしない関数だとこのアプローチは不利である．

3.3.2　演算子 “=”

代入演算子 “=” は多相型をもつオブジェクトに対しても頻繁に使われる．この演算子もユーザー定義の構造体やクラスに対して多重定義することができる．もし，演算子 “=” を他の関数と同様に関数の引数として渡すことにすると “=” の boxing，unboxing の変換，関数呼び出しのオーバーヘッドは莫大になる．

演算子 “=” を型に依らずに一語のコピーに変換すると，“=” に対して boxing，unboxing，関数呼び出しを行う必要はなくなる．

こうすると，変換前のソースでは値をコピーであったものが，ポインタのコピーになり，同じ領域を指すポイ

ンタが新たに発生しうる．もとのコードにはなかったポインタ管理が必要となる．

まず，演算子 “.” による型多相なオブジェクトの更新は，他の変数が同じ構造体を指しているかもしれないので，構造体をヒープの別領域にコピーして，そのコピーに対して変更を行わなければならない．型多相な構造体をヒープ上に確保する関数（new に当たるもの）は多重定義された関数同様に追加の引数として受け取らなければならない．

さらに，型多相な値は他に参照している変数があるかも知れないので，値を指す変数が全て使われなくなるまでメモリ領域を解放できない．逆に，どの変数からも参照されなくなった値は解放しなければならない．ある値を参照する変数がいくつあるかは動的に変化するので，この方法を使うときにはガーベージコレクタが必要である

4. その他のテンプレートの機能

4.1 明示的特別化

テンプレート関数は，具体的な型に対して特別化した定義をあたえて，そちらを優先して適用することができる．例えば次のような例が考えられる．

```
template <class T>
void print(T x) {
  cout << "template varsion called"
       << endl;
}

template <>
void print(double x) {
  cout << "double version called"
       << endl;
}

template <class T>
void f(T x) {
  print(x);
}

void foo() {
  //call template version
  f(1);
  //call double version
  //(specialized)
  f(1.9);
}
```

この例では “f(1.9)” の呼び出しでは，“print(double)” の方が呼び出され，“double version called” と出力される．

この例の関数 “f” は，多相型 T に対して定義された “print” 関数を適用側から受け取るように変換される．int に特別化された print を渡すか，テンプレート関数で定義された print 関数を渡すかは適用側の変換時に引数の型情報から決める．テンプレート関数の “print” が呼ばれる場合は，引数は boxing されたデータを受け取るので，3.1.2で説明したような，boxing された引数を unboxing して “print” を呼ぶ関数は定義しなくて良い．変換後のソースは以下のようになる．

```
//# template <class T>
//# void print(T x)
void print(wrapped_t x) {
  cout << "Template varsion called"
       << endl;
}

//# template <>
void print(double x) {
  cout << "double version called"
       << endl;
}

//# template <class T>
//# void f(T x)
//# require void print(T)
void f(T x) {
  print(x);
}

void print_on_double(wrapped_t x) {
  double dx = UNWRAP(x);
  print(dx);
}

void foo() {
  //call template function
  wrapped_t tmp = (wrapped_t)1;
  void (*tmp2)(wrapped_t) = print;
  f(tmp,tmp2);

  //call double version
  //(specialized)
  wrapped_t tmp3 = WRAP(1.9);
```

```
  f(tmp3,print_on_double);
}
```

4.2 継承関係の親をパラメタ化する場合

もし親クラスがテンプレートでパラメタ化されていた場合，その子クラスは親の public または protected なメンバにアクセス可能である．

```
template <class Parent>
class Child : public Parent {
  void goo();
  void print();
};

template <class Parent>
void Child<Parent>::goo() {
 //access to parent's member
 //(public or protected)
  Parent::foo();
}

template <class Parent>
void Child<Parent>::print() {
  cout << "I'm child" << endl;
}
```

この例のような場合，このクラステンプレートをコンパイルするときには，親クラスが何であるかは分からない．よって，このテンプレートの型引数に具体的な型を代入して使用される箇所を変換する際に，実際の親クラスを継承するクラスを定義する必要がある．そのクラスで，子クラス “Child” で実装が定義されているものについては，それを呼び出すコードを変換によって挿入する．

```
class Father {
protected:
  void foo() { //... }
};
typedef Child<Father> Son;
//# template <class Parent>
//# class Child : Parent
//# require Parent::foo()
class Child {
  void zoo() {
    //...
  }
}

class Son : public Father {
  //private:
  //  void foo() { //... }
public:
  //Child::zoo is shared
  void goo() { Child::goo(); }
  void print() { Child::print(); }
}
```

この場合はテンプレートで定義されたクラス “Child” の “print” のような，オーバーライドされたコードや，親にはなく新たに定義されたメソッドのコードが，共有される．

テンプレート関数からは public か protected なメンバのみを見ることができ， private なメンバは見ることができない．なぜなら，テンプレート関数の型引数はたった今定義しようとしているクラスの名前をパラメタ化することはできないからである．つまり private なメンバについてのアクセスの可能性が変換後に代わることはない．

4.3 考慮していない機能

本論文のアプローチは traits[2] や value template に対応するかどうかは考慮していない．たとえば， traits を使うと次のような構造体を定義できる．

```
template <class T>
struct p {
  T first;
  T second;
};

struct p<double> {
  double first;
  double second;
  double third;
};

template <class T>
T get_first(p<T> x)
{
  return x.first;
}
```

本来のテンプレートでは，関数 “get_first” は，テンプレートで定義された構造体 p も double 型で特別化された構造体 p もどちらも扱うことができる．このような場合に対応するには本論文の方法を更に拡張する必要がありそうである．

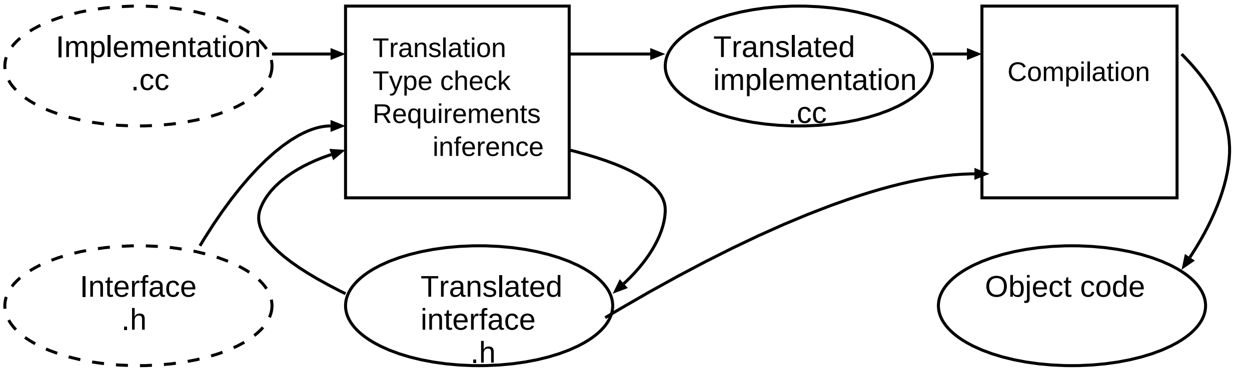

図 9　変換とコンパイルの流れ

5. 以上を考慮した変換器の構成

5.1　コンパイルの過程

本論文が提案した変換とコンパイルは図 9に描かれているように行われる．最初に，変換器が実装を含むソースファイルと宣言を含むヘッダファイルを読み込む．変換器は前節までで述べた変換でテンプレートを取り除き，型多相なオブジェクトに適用している関数についての情報を集める．そして，変換されたソースコードとヘッダファイルを出力する．変換後のヘッダファイルは前節までで述べた多相型に適用する関数についての情報を含む．

この変換によって，テンプレート関数で不足していた多相型についての情報が関数の宣言として現われる．例えば次の “max” 関数の変換について見てみる．従来のテンプレート関数の宣言は以下のようになる．

```
template <class T>
T max(T x,T y);
```

この宣言は引数の x,y の型と関数の返り値の型が全て同じであることを要求しているにすぎない．この “max” 関数が x,y に演算子 > を適用していた場合，変換後の関数宣言は図 4の関数 “max” ようになり，T 型に対して演算子 > が必要であるという情報が現われる．

この “greater_than” は多重定義を解決するために導入された引数であるが，テンプレートの多相型に対する型情報を補う役目も果たしている．この情報は “max” の呼び出しを変換する際に必要である．さらに “max” を利用するプログラマに取っても必要な情報である．

5.2　依存性の取り扱い

本論文の変換ではファイルの依存は大きく分けて 2 種類考えられる．

多相なオブジェクトに対する要求を変換器によって自動的に抽出させると，変換によって自動的に宣言が得られる半面，多相関数の実装が宣言とその使用に依存するため，実装がなければそれを使うコードのチェックができないという欠点がある．

もう 1 つは変換後の宣言にあたるものを予めプログラマーが書いて，適用側も多相定義の実装側もその宣言に従うという方法である．この方法では，多相関数の定義も，その使用も宣言に依存するので，実装がない状況で，適用側が定義側の要求を満たすかをチェックできる．

この両方を折衷して，完全ではないが多相型に対する情報を抽出するのに充分な実装を変換して，インターフェイスを自動生成し，以後はそのインターフェイスに実装側，適用側が従うという方法も考えられる．

6. 実 験 結 果

テンプレートを使用したプログラムを手で変換して実行効率と実行可能ファイルのサイズを調べた．以下すべての実験は PentiumII 450Mhz，メインメモリ 128MByte のマシンで行った．時間は gettimeofday 関数を用いて計測した． C++ のコンパイルには GNU C++ compiler のバージョン 3.02 を最適化レベル 3 で使った．すべての実行ファイルは strip を使ってシンボル情報を取り除いている．この実験での変換後のコードは全て 3.3.2節で述べたように，演算子 “=” は多重定義されていないものと仮定して扱っている．

1 つめのプログラムは要素が多相型であるクイックソートである．この実験では double の配列をソートした．この場合クイックソートは要素に適用する演算子 “<”，“>” と配列参照の演算子 “[ ]” を必要とする．この演算子を 3.3.1で述べた二通りのアプローチを使って，従来のテンプレートのコンパイル方法でコンパイルしたものと実行効率，コードサイズを比較した． 2. の予め各要素を boxing した配列を渡すアプローチは配列の長

表 **1** クイックソートの実行効率

	実行時間 (秒)
1. 変換なし	0.023
2. 配列要素を予め boxing	0.086
3. 配列要素をアクセスの度に boxing	0.57

表 **2** クイックソートの実行可能コードのサイズ (byte)

	double	double & int	double & int & char
1. 変換無し	4376	4576	4772
2. 配列要素をを予め boxing	4740	4964	5124
3. 配列要素をアクセスの度に boxing	5344	5608	5848

さに比例した boxing，unboxing のオーバーヘッドを必要とする．一方で 3. の配列アクセスの度に boxing，unboxing を行うアプローチでは配列長の対数オーダーの boxing，unboxing を必要とする．

3. は従来のアプローチ 1. と比べて 20 倍以上実行時間がかかっている．2. のアプローチを使うことで従来の方法の 3.5 倍程度に抑えられる．この結果から，ボックス化，unboxing のオーバーヘッドは大きく，何らかの工夫をすることで，実行効率の損失を抑えられることが分かる． コードサイズを比較すると，本論文のアプローチを用いてコンパイルされたコードは従来のコンパイル方法によるものに比べてかえってコードが大きくなってしまっている．これは，quick_sort 関数自体のコードに比べて，変換によって新たに加えられる関数のコードのサイズが大きいためである．

2 つめのテストは要素が多相型である集合演算のプログラムである．functor を用いて実装されている Ocaml[3] のライブラリを C++ のテンプレートを用いて移植し，それを手で変換した．集合の要素に必要な関数は，2 要素間を比較する関数 compare である．この集合は実際はバランスした 2 分木で実装されているが，その実装は隠蔽されており，集合に対して決められた関数を通してのみアクセスできるようになっている．逆に集合に対して決められた関数からアクセスできればそれで良いので，集合の要素は常に boxing された状態で集合内に存在している．つまり，集合を再帰的に boxing，unboxing はしていない．

集合演算の実行効率は表 3に示した通りである．このプログラムではdouble 型のデータに対して集合の主な操作を一通り行う．add，remove，filter，fold，cardinal（集合内の要素の数を数える），そして，max_elt，min_elt をテストしている．表を見るとクイックソートのプログラムほどは実行効率が落ちていないことが分かる．要素型が求める演算が compare 関数一つのみであり，compare 関数を引数として渡すオーバーヘッドや compare 関数内での boxing，ボックス取り出し化のオーバーヘッドが顕著に現われていないためだと考えられる．

表 4 は集合演算の適用する型に対するコードサイズの増加の様子を示している．quick_sort と異なり，集合演算はバランスした木を訪問したり，木のノードを扱うという要素型に直接関係しない操作が多いので，変換後のコードは異なる型の間でのコード共有がより進んでいる．このようなプログラムに対しては本論文の方法は有効である．

7. 関 連 研 究

Veldhuizen[4] は C++ の 5 つのコンパイルモデルを提案した．そのなかのいくつかはテンプレートの定義をその適用と分割コンパイル可能にするものである．C++ は中間言語に変換される．その中間言語ではそれぞれの型を “type_info” という構造体で表現し，テンプレート関数の型変数は実行時に渡される．この型情報は多重定義された関数の解決に使われる．本論文の方法は，実行時に型情報を受渡しするのではなく適用のコンパイル時に解決している．

Leroy[5] は ML で boxing されたオブジェクトと，ボックスされていないオブジェクトを混合した表現を使う方法を提案した．ほとんどの ML コンパイラは多語のデータに boxing した表現を用いるが，プログラムの型単相な部分は不必要な boxing，unboxing で効率が低下する．そこで，Leroy は型単相なコードは boxing されていないオブジェクトを直接扱う変換をすることで，実行効率を上げることを目指した．本論文では同様な方法を C++ のテンプレートの分割コンパイルを目的として使用した．また，この boxing，unboxing の操作を，多重定義された関数に対しても適用した．

Wadler[6] はアドホック多相性を解決する “type class” を提案した．Wadler の方法では，多重定義はあるクラスに多重定義された関数が登録されている辞書を関数に渡すことで解決している．本論文の多重定義に対するアプローチはこの論文の方法を参考にしている．本論文で

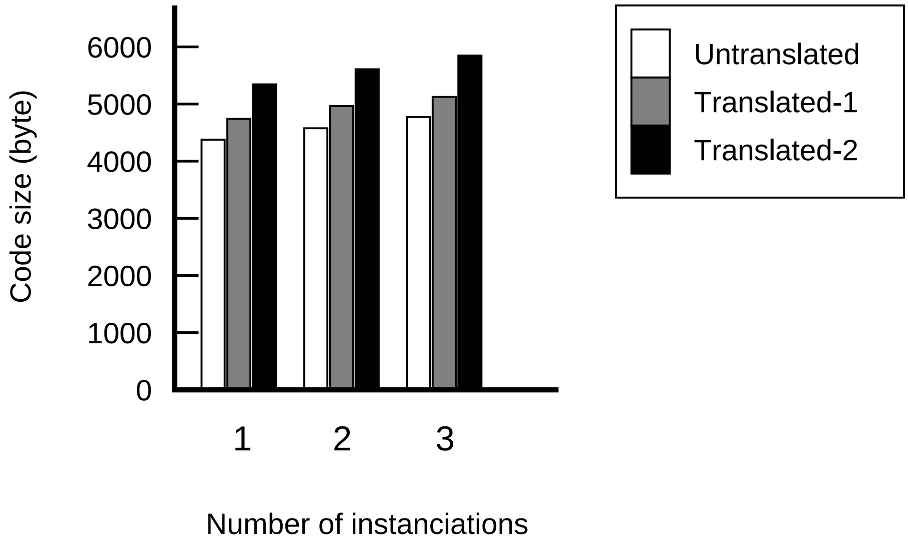

図 **10**　クイックソートの実行可能コードのコードサイズ
Translated-1 配列要素を予め boxing
Translated-2 配列要素をアクセスの度に boxing

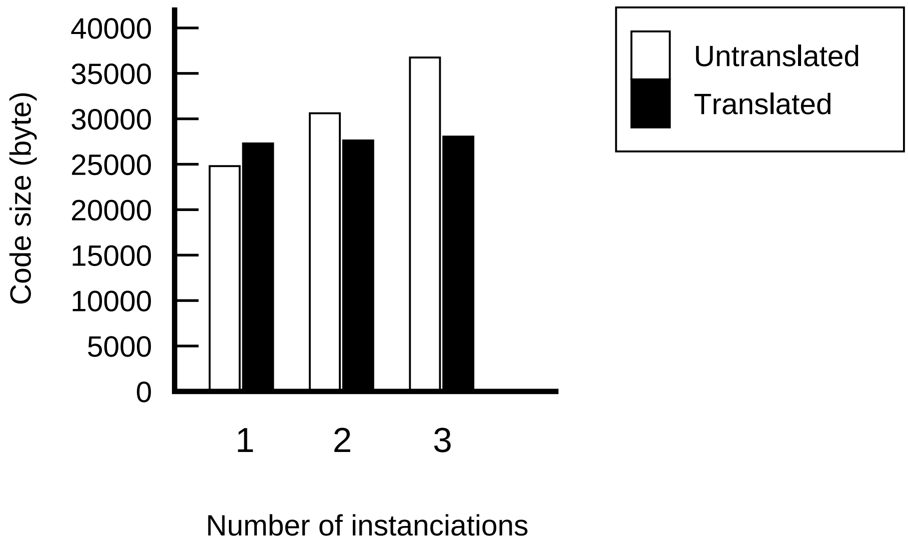

図 **11**　集合演算の実行可能コードのコードサイズ

表 3 集合演算の実行効率

	実行時間 (秒)
変換なし	0.14
変換あり	0.17

表 4 集合演算の実行可能コードのサイズ (byte)

	char*	char* & int	char* & int & double
変換なし	24796	30612	36748
変換あり	27292	27620	28044

は “辞書” を渡すのではなくて，関数そのものを渡す点が異なる．

Elsman[7)] は同じ型の 2 引数をとる型多相な等価性 を調べる関数について述べている。この論文はペアの個々の要素に対する等価性を調べる関数は関数の引数として渡される。本論文では，この方法を拡張して関数は任意の数の引数，任意の型をとりうる多重定義された関数を扱った。

MLton ，MLj[8)] はほとんどの C++ コンパイラと同様に，コンパイル時に多相型を単相化するアプローチを採用している．MLton は Standard ML のソースコードを一括コンパイルする．MLj は Monadic 中間言語から基本ブロックコード (BBC) に変換する際にこのアプローチを用いている．両者は多相型を一括コンパイルすることで単相化し実行効率を上げることを目的としている．一方で，両者ともコンパイル時間が多くかかってしまうことや、コードサイズの肥大化することが問題になっている．

8. 結論と今後の課題

本論文では C++ のテンプレート関数の定義とその適用を分割コンパイルするアプローチを提案した．このアプローチは，従来のコンパイル方法で問題になっていたコードサイズの肥大化を抑えうる．

また，多相型に対して多重定義の制約を記述できるようにして，演算子の多重定義を制限すると，実行効率のオーバーヘッドを許容できるほどに抑えながら，テンプレートの分割コンパイルを実現できる．

traits,value template などの本論文で考慮されていないテンプレートの機能については今後考慮したい．

謝辞 数々の助言，励ましを頂いた研究室関係者の方々に謹んで感謝の意を表する．

参 考 文 献

1) Stepanov, A. A. and Lee, M.: The Standard Template Library, Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project (1994).

2) Veldhuizen, T. L.: C++ Templates as Partial Evaluation, *Partial Evaluation and Semantic-Based Program Manipulation*, pp.13–18 (1999).

3) Leroy, X.: The Objective Caml system: Documentation and user's guide (1996).

4) Veldhuizen, T. L.: Five compilation models for C++ templates, *First Workshop on C++ Template Programming, Erfurt, Germany* (2000).

5) Leroy, X.: Unboxed objects and polymorphic typing, *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992* (ACM(ed.)), New York, NY, USA, ACM Press, pp. 177–188 (1992).

6) Wadler, P. and Blott, S.: How to make *ad-hoc* polymorphism less *ad hoc*, *16th ACM Symposium on Principles of Programming Languages*, pp. 60–76 (1989).

7) Elsman, M.: Polymorphic Equality – No Tags Required, *Second International Workshop on Types in Compilation*, Springer Verlag (1998). to appear.

8) Benton, N., Kennedy, A. and Russell, G.: Compiling Standard ML to Java Bytecodes, *ACM SIGPLAN Notices*, Vol. 34, No. 1, pp. 129–140 (1999).

(平成?年?月?日受付)
(平成?年?月?日採録)

増山 隆

1979 年生．2002 年東京大学理学部情報科学科卒業．現在，東京大学理工学系研究化コンピュータ科学専攻修士過程に在学．特にモジュールシステムとコードの再利用に関して興味を持っている．

住井英二郎

1975年生．2000年3月東京大学大学院理学系研究科情報科学専攻修士課程修了．同年4月博士課程進学．同年より翌年にかけて，米国ペンシルバニア大学訪問研究者．2001年4月より東京大学大学院情報学環助手 (情報理工学系研究科コンピュータ科学専攻より流動)．関数型言語，プロセス計算，部分評価，情報セキュリティ等の領域における，先進的な型システムの理論と応用に関する研究に従事．ACM会員．

米澤　明憲（正会員）

1947年生．1977年 Ph.D. in Computer Science (MIT)．1989年より東京大学理学部情報科学科教授．超並列・分散ソフトウェアアーキテクチャなどに興味を持つ．共著書「モデルと表現」等 (岩波書店)、編著書「ABCL」(MIT Press) 等がある．1992–1996年ドイツ国立情報処理研究所 (GMD) 科学顧問、ACM Transaction on Programming Languages and Systems 副編集長、IEEE Parallel & Distributed Technology および Computer 編集委員などを歴任、元ソフトウェア科学会理事長、ACM Fellow．