

The Boost C++ Metaprogramming Library

Aleksey Gurtovoy
MetaCommunications
agurtovoy@meta-comm.com

David Abrahams
Boost Consulting
david.abrahams@rcn.com

概要

この文書は、アルゴリズム、シーケンス、メタ関数クラスの拡張可能なコンパイル時フレームワークである、**Boost C++ テンプレートメタプログラミングライブラリ (MPL)** について述べている。ライブラリは実環境での実際の実際の仕様に十分な、強力で使いやすいツールセットを構築するために、ジェネリックプログラミング、及び関数プログラミングの世界から、重要な抽象化を導入している。MPL は、C++標準ライブラリ [STL94], [ISO98]の一部である、標準テンプレートライブラリ (STL) の影響を強く受けている。STLの様に、MPLはこの領域に将来貢献するための土台となるような、公開された概念と実装の枠組みを定義している。ライブラリの基本的な概念とイディオムにより、ユーザが与えられたメタプログラミングの問題に対して、アドホックなアプローチの世界に引きづり込まれることなく、解決へ向かうことができるようになる。たとえ、実際のMPLのコードが使われなくても、である。ライブラリはまた、コンパイル時ラムダ式的能力を提供する。これは、任意のカリエー化とクラステンプレートの合成を可能にする。この特徴は、ランタイム時の対応物として引用される STLには存在しない。この文書は、MPLの動機、使用法、デザイン、実装について、実際の応用例と共に説明し、C++テンプレートメタプログラミングについて学んできた教訓を提供する。

目次

1. イントロダクション	3
1.1. ネイティブ言語によるメタプログラミング	3
1.2. C++ におけるメタプログラミング	3
1.2.1. 数値計算	3
1.2.2. 型計算	4
1.2.3. 型シーケンス	4
1.3. なぜメタプログラミング?	6
1.4. なぜメタプログラミングライブラリ?	8
2. 基本的な利用法	8
2.1. 条件付き型選択	8
2.1.1. 遅延評価	9
2.2. メタ関数	12
2.2.1. 単純な形式	12
2.2.2. 高次のメタ関数	12
2.2.3. メタ関数クラス	14
2.2.4. ひとつで十分?	14
2.2.5. メタ関数からメタ関数クラスへ	15
2.3. シーケンス, アルゴリズム, イテレータ	15
2.3.1. イントロダクション	15
2.3.2. アルゴリズムとシーケンス	16
2.3.3. シーケンスコンセプト	16
2.3.4. アドホックな例, 再び	18
2.3.5. 主要反復アルゴリズムとしてのiter_fold	18
2.3.6. 数値シーケンス	19
2.3.7. いろいろなシーケンス	20
2.3.8. ループ/再帰の展開	21
3. ラムダ機構	23
4. コード生成機構	25
5. 例: コンパイル時 FSM ジェネレータ	26
5.1. 実装	28
5.2. 関連する研究	30
6. 謝辞	30
References	31

1. イントロダクション

メタプログラミングは通常、別のプログラムを生成するプログラムを作成することとして定義される。YACC[Joh79]のようなパーサジェネレータは、一種のプログラム生成プログラムである。YACCへの入力言語は拡張BNF[EBNF]の文脈自由分布であり、出力はその文法を解析したプログラムである。この場合、メタプログラミング(YACC)は、生成されるプログラムの記述を直接サポートしない言語(C)で書かれている。metadataと呼ばれるこれらの仕様は、Cで書かれているわけではなく、メタ言語で書かれている。ユーザのプログラムの残りは通常、汎目的なプログラミングシステムを要求し、生成されたパーサとやりとりしなければならない。そうすることで、metadataはCに翻訳され、それから残りのシステムと共にコンパイル、リンクされるのである。metadataはこのように、2つの翻訳段階を踏み、ユーザは常に、metadataとプログラムの残りの境界線を意識しているのである。

1.1. ネイティブ言語によるメタプログラミング

metaprogrammingにおいてもっと興味深い形式は、Scheme[SS75]の様な言語で使うことができる。Schemeでは、生成されたプログラム仕様は、メタプログラムとして同一言語の中で与えられる。メタプログラマは言語の中で拡張可能な形式のサブセットとしてメタ言語を定義し、プログラム生成はユーザプログラムの残りの過程に使われるのと同じ翻訳段階で行われる。これにより、ユーザは透過的に通常のプログラミング、生成されたプログラムしよう、そしてmetaprogrammingを切り替えられるし、大抵、その移行には気づかないのである。

1.2. C++ におけるメタプログラミング

C++では、テンプレート機構がコンパイル時計算に対して豊富な機能を提供する、ということがほとんど偶然[Unr], [Ve195a]発見された。この章では、C++におけるメタプログラミングで使われる基本的な機構といくつかのイディオムを探ってみる。

1.2.1. 数値計算

非型テンプレートパラメータの有用性は、コンパイル時整数演算を可能にする。例えば、以下のテンプレートはその引数の階乗を計算する：

```
template< unsigned n >
struct factorial
{
    static const unsigned value = n * factorial<n-1>::value;
};

template<>
struct factorial<0>
{
    static const unsigned value = 1;
};
```

上のようなプログラムの断片化はmetafunctionと呼ばれ、実行時に評価されるように設計された関数との関係を見て取るのは容易である：「metafunction 引数」はテンプレートパラメータとして渡され、「返り値」はネストした静的定数(static constant)として定義される。C++でのコンパイル時式と実行時計算の間の大きな隔たり(?)のため、メタプログラミングはそれと対応する実行時計算とは異なって見える。このように、Schemeと同じくC++メタプログラマは通常のプログラムと同一の言語でコードを書くことができるが、その際、完全なC++言語のサブセットが使えるに過ぎない：それらはコンパイル時に評価されることが可能な式である。上のコードを直接、実行時の階乗関数の定義と比べれば：

```
unsigned factorial(unsigned N)
```

```
{
    return N == 0 ? 1 : N * factorial(N - 1);
}
```

2つの再帰定義の間のアナロジーを見て取るのは容易だが、再帰は通常、実行時C++よりもC++メタプログラミングにとって、より重要である。再帰が言語独特であるLispのような言語と比べ、C++プログラムは通常、出来る限り再帰を避ける。これは効率という理由だけでなく、「文化的要因」も大きい：再帰プログラムは単純に(C++プログラマにとって)考えるのが難しい。しかし純粋なLispの様に、C++テンプレート機構は関数的プログラム言語である：ループ変数を維持するために必要なデータ操作を使うことはしないのである。

実行時とコンパイル時の階乗関数の主要な違いは、終了条件式である：メタ階乗はNがゼロの時のビヘイビアを記述するために、パターンマッチング機構の一種としてテンプレート特殊化を使っている。実行時世界との構文的類似物は、同じ関数の2つの別の定義を要求する。この場合、2番目の定義のインパクトは最小であるが、大きなメタプログラムでは、終了定義を維持し理解するコストは重要になる。

C++ metafunction の返り値は名前付けされていなければならないことに注意。ここで選ばれた名前、**value** はMPLで全ての数値的な返り値に使われているものと同じである。これから見るように、metafunctionの返り値にとって、一定の名前付けの習慣を確立することは、ライブラリの力に対して決定的である。

1.2.2. 型計算

factorial metafunctionをどうやって応用すればよいのだろうか。例えば、異なる型の実体の全ての順列を保持するのに適した大きさの配列型を作りたいとする。

// permutation_holder<T>::type は配列型。これは与えられたTの全ての順列を含むことができる。

```
// スカラのための非特殊化版テンプレート
template< typename T >
struct permutation_holder
{
    typedef T type[1][1];
};

// 配列型のための特殊化版
template< typename T, unsigned N >
struct permutation_holder<T[N]>
{
    typedef T type[factorial<N>::value][N];
};
```

ここで型計算という概念を導入した。上の**factorial**の様に、**permutation_holder**テンプレートはmetafunctionである。しかし、**factorial**が符号なし整数値を扱うのに対し、**permutation_holder**は(ネストした**typedef type**として)型を受け取り、「返す」。C++ 型システムは非型テンプレート引数(例えば整数)として使う式より遙かに豊富な式集合を提供するので、C++メタプログラミングは、ほとんど型計算の合成となる傾向がある。

1.2.3. 型シーケンス

プログラムが型集合を扱う能力は興味深いC++メタプログラミングの中でも中核に位置する。この能力は、MPLによりよくサポートされているので、ここでは基本的なことを少しだけ紹介しておく。後に、MPLを使ってどのように実装できるかを示すために、下の例をもう一度扱うだろう。

まず、集合を表現する方法が必要である。ひとつのアイデアは、構造体に型を保持する、というものだ。

```
struct types
```

```
{
    int t1;
    long t2;
    std::vector<double> t3;
};
```

不幸なことに、このやり方は、C++が提供するコンパイル時型チェックの力を損なう：メンバの名前が何であるか見つける方法はないし、たとえ既に述べたような習慣に倣って名前付けしていると仮定しても、そこにどれくらい多くのメンバがあるのか知る方法はない。この問題をとくための鍵は、表現の同一性を増やすことである。もしどんなシーケンスであっても最初の型を得ることができて、さらにシーケンスの残りを得ることができるような、一定の方法があれば、簡単に全てのメンバにアクセスできる：

```
template< typename First, typename Rest >
struct cons
{
    typedef First first;
    typedef Rest rest;
};

struct nil {};

typedef
    cons<int
    , cons<long
    , cons<std::vector<double>
    , nil
    > > my_types;
```

上の**types**で記述された構造体は単一リンクリストのコンパイル時類似物である。それはCzarneckiとEiseneckerによって、初めて導入された[CE98]。今や、構造体を調整したので、C++テンプレート機構はそれを「加工する」ことができる。それを行う単純なメタ関数を試してみよう。ユーザは任意の型集合から、最も大きなものを見つけないとする。ここで、すでに親しんでいる再帰的metafunction形式を応用できる：

例 1. '最大' メタ関数

```
// 2つの型のうち大きい方を選ぶ
template<
    typename T1
    , typename T2
    , bool choose1 = (sizeof(T1) > sizeof(T2)) // 渡す!
    >
struct choose_larger
{
    typedef T1 type;
};

// sizeof(T2) >= sizeof(T1) の場合の特殊化版
template< typename T1, typename T2 >
struct choose_larger< T1, T2, false >
{
    typedef T2 type;
};

// コンスリストから最大のものを得る
template< typename T > struct largest;

// コンスリストから引きはがすための特殊化版
template< typename First, typename Rest >
struct largest< cons<First, Rest> >
```

```

    : choose_larger< First, typename largest<Rest>::type >
{
    // baseを継承した型
};

// ループ終了のための特殊化版
template< typename First >
struct largest< cons<First,nil> >
{
    typedef First type;
};

int main()
{
    // my_typesのうち最も大きなものを出力
    std::cout
        << typeid(largest<my_types>::type).name()
        << std::endl
        ;
}

```

このコードについて書いておく価値のあることがいくつかある。

- いくつかのアドホックで、深遠な技、というか「ハック」を使っている。デフォルトテンプレート引数`choose1`(「渡す!」とラベルされたところ)がひとつの例となろう。これがなければ、`choose_larger`の実装を提供する、また別のテンプレートを必要とするか、テンプレートへのパラメータとして、明示的に計算を与える必要があるだろう。それもおそらくこの例では悪くないだろうが、`choose_larger`は余り役立たなくなるし、エラーを生みやすくなる。別のハックは`choose_larger`から`largest`の特殊化版を派生しているところだ。これは、プログラマが「`typedef typename ...::type type`」とテンプレートの中で書くのを避けるという点で、省コードなものである。
- この単純なメタプログラムでも、3つの異なる部分特殊化を使っている。`largest` metafunctionは2つの特殊化を使っている。これに対して、2つの終了条件があることを示しているとも考えられるかもしれないが、そうではない: ひとつの特殊化は単純にシーケンスの要素へのアクセスを扱うために必要なだけである。これらの特殊化は、ひとつのmetafunctionを多くのC++テンプレート定義へと分散してしまうことで、コードを読みにくくする。また、それらは部分的特殊化なので、この特性をサポートしないコンパイラを使っているC++プログラマの人々にとって、使えないコードということになる。

これらのテクニックは勿論、多くの優れたC++メタプログラマの蓄積の価値ある一部であるが、これらを使うと、プログラムは、読みづらく書きづらい、異常なスタイルで書かれることになる。広く使われる構造をカプセル化し、ループ終了を内的に扱うことで、MPLはトリッキーなハックとテンプレート特殊化の両方の必要を軽減している。

1.3. なぜメタプログラミング?

なぜこれを行いたい者がいるのか尋ねるのは価値あることである。結局、階乗 metafunction のような単純なお遊びの例でも、ある程度深遠なものである。型計算がどのように機能するのかわかるために、単純な例を試してみよう。次のコードは別の配列のすべての可能な順列を含む配列を生成する。

```

// C++の配列は返すことは出来ないので、このラッパーが必要である。
template< typename T >
struct wrapper
{
    T x;
};

```

```

// 'in' の N!の順列の配列を返す。
template< typename T >
wrapper< typename permutation_holder<T>::type >
all_permutations(T const& in)
{
    wrapper<typename permutation_holder<T>::type> result;

    // 順列ではない配列を最初の結果の要素にコピーする
    unsigned const N = sizeof(T) / sizeof(**result.x);
    std::copy(&*in, &*in + N, result.x[0]);

    // 順列を列挙する
    unsigned const result_size = sizeof(result.x) / sizeof(T);
    for (T* dst = result.x + 1; dst != result.x + result_size; ++dst)
    {
        T* src = dst - 1;
        std::copy(*src, *src + N, *dst);
        std::next_permutation(*dst, *dst + N);
    }
    return result;
}

```

factorialの実行時定義は、上のall_permutationsの中で使うことは出来ない。C++では配列メンバの大きさはコンパイル時に計算されなければならないからである。しかし、別のアプローチがある。どのように metaprogramming を避けることが出来て、その結果どうなるのだろうか？

1. metadataを直接解釈するプログラムを書くことが出来る。階乗の例では、配列の大きさは実行時に決めることが出来る。それから単純な階乗関数を使うことが出来る。しかし、たいいてい無駄なコストがかかる動的メモリ割り当てを使うことになる。

これをさらに進めれば、YACCはパースされるストリームから、トークンを返す関数ポインタを受け取り、また文法記述を含む文字列を受け取るように書き直すことが出来る。しかし、このアプローチでは、多くのアプリケーションにとって許容外の実行時コストがかかってしまう：パーサはパースのたびに文法を探索しながら、非決定的に文法を扱うか、実行時に文法入力のために、存在するYACCの従属テーブル生成と最適化を、再度行うことから始めることになる。

2. われわれの独自の分析によれば、コンパイル時計算で置き換えることが可能である。結局、all_permutationsに渡される配列の大きさは、常にコンパイル時に知られていて、ユーザも知ることが可能である。そして、ユーザが結果型を明示的に提供するように求めることが可能である：

```

template< typename Result, typename T >
Result all_permutations(T const& input);

```

このアプローチのコストは明白だ：(ユーザが明示的に実装詳細を指定するように要求することにより) 表現性を失い、(ユーザがそれらを誤って指定することが可能なので) 正しさを失う。パーサテーブルを手で書いたことがあるならば、このアプローチが実践的でないことがまさに、YACCが存在する理由なのだ、と言うかもしれない。

C++のような言語では、metadata がユーザのプログラムの残りと同じ言語で表現されるなら、表現性はかなり改善される：ユーザはメタプログラムを直接呼び出すことが出来て、コードのフローを邪魔する、異なる構文を学ぶ必要はない。

結局、metaprogrammingの動機は3つの要因の組み合わせとすることが出来る：効率、表現性、正当性である。クラシックなプログラミングでは常に表現性と正当性、表現性と効率はトレードオフであったが、metaprogrammingでは新しい力を振りかざすことが出来る：表現性のために、必要な計算を実行時からコンパイル時に移すことが出来るのだ。

1.4. なぜメタプログラミングライブラリ?

なぜ汎用ライブラリが必要なのか、とたずねるものもいるだろう:

- ・ 質。汎目的なライブラリに妥当なコードは、通常そのユーザの目的にとって不必要である。ライブラリ開発者にとって、それは中心的な使命である。平均的に、C++標準テンプレートライブラリにより提供されるコンテナとアルゴリズムは、よくあるプロジェクト独自の実装に比べて、より柔軟だし、うまく実装されている。ライブラリ開発は、他のアプリケーションの開発には不必要な課題ではなく、むしろそれ自身が目的として扱われるのだから。どんな関数でも、その中心となる実装に対して、最適化と改善は適用されるのである。
- ・ 再利用性。すべてのライブラリが提供する、コードの再利用性よりもさらに重要なことだが、よく設計された汎用ライブラリは、問題にアプローチするための再利用可能なメンタルモデルを確立する。概念とイディオムの枠組みを確立する。C++標準テンプレートライブラリが、イテレータの概念と関数オブジェクトプロトコルを与えてくれたように、Boost Metaprogramming Libraryは型イテレータとメタ関数クラスプロトコルを提供する。よく考えられたイディオムの枠組みにより、メタプログラミングは無関係な実装詳細を考える必要がなくなり、大事な問題に集中することが出来るようになる。
- ・ 移植性。よいライブラリはプラットフォームの違いという醜い現実を容易に乗り越える。理論的にはmetaprogrammingライブラリは完全に汎用的でこの問題に関わるべきではないのだが、実際はテンプレートのサポートは標準化から4年たってなお、一致していない。これは驚くべきことではない: C++テンプレートはこの言語の最も遠いゴールであり、最も複雑な特徴である。C++では、metaprogrammingの力の大部分はテンプレートによるのである。
- ・ 楽しさ。同じイディオムを何度も何度も繰り返すことは、退屈だ。プログラマは疲れるし、生産性は減少する。さらに、飽きて不注意になり、バグ混じりのコードはゆっくり書かれたコードよりもコストがかかる。たいてい、最も使えるライブラリというのは、抜け目ないプログラマを繰り返しの海から「救う」単純なパターンである。MPLは多くの広く繰り返される、決まり文句的なコードのパターンの必要をなくすことで、退屈さを軽減する。

見て解るように、MPLの開発の動機は、基本的には他のライブラリの開発と同じように、実践的な、実世界の考慮による。おそらくこれは、テンプレートmetaprogrammingが最終的に、深遠な領域を脱して、日常プログラマのリングフランカ になろうとしてることの徴候である。

2. 基本的な利用法

2.1. 条件付き型選択

条件付き型選択はC++テンプレートメタプログラミングのもっとも基本的な構築となる。Veldhuizen [Vel95a] はこれをどのように実装するか最初に示した。Czarnecki と Eisenecker [CE00] はスタンドアロンのライブラリ プリミティブとしてこれを最初に提出した。MPLは以下のものと同等の能力を定義する。

```
template<
    typename Condition
    , typename T1
    , typename T2
>
struct if_
{
    typedef /*unspecified*/ type;
};
```

テンプレートの最初のテンプレートパラメータが型であることに注意。

```
// usage/semantics
typedef mpl::if_<mpl::true_c, char, long>::type t1;
typedef mpl::if_<mpl::false_c, char, long>::type t2;
```

```
BOOST_MPL_ASSERT_IS_SAME(t1, char);
BOOST_MPL_ASSERT_IS_SAME(t2, long);
```

テンプレートメタプログラミングは多くの意思決定コードを含み、またこれから示すように、それを(部分的)クラステンプレート特殊化によって毎回手書きするのはすぐに非現実的になるので、構築は重要である。テンプレートはまた、コンパイラのワークアラウンドをカプセル化するという点からも重要である。

2.1.1. 遅延評価

C++ テンプレートインスタンス化機構が機能する方法により、型選択プリミティブ(`if_`)の応用可能性には、同等の選択コードを手書きするのに比べて多少の制限が加わる。例えば、生ポインタ(`U*`)か、`std::auto_ptr<U>`か、あるいはBoostのスマートポインタ[SPL]、例えば`boost::scoped_ptr<U>`のいずれかであるTに対してインスタンス化され、ポインタ型(U)を返す、`pointed_type<T>::type`のような、`pointed_type`特性テンプレートを実装することを考えてみよう。

```
BOOST_MPL_ASSERT_IS_SAME(pointed_type<my*>::type, my);
BOOST_MPL_ASSERT_IS_SAME(pointed_type< std::auto_ptr<my> >::type, my);
BOOST_MPL_ASSERT_IS_SAME(pointed_type< boost::scoped_ptr<my> >::type, my);
```

不幸なことに、この問題に対して`if_`を単純に応用してもうまくいかない：¹

```
template< typename T >
struct pointed_type
    : mpl::if_<
        boost::is_pointer<T>
        , typename boost::remove_pointer<T>::type
        , typename T::element_type // #1
    >
{
};
```

```
// 次のコードは#1の行でコンパイルエラーを起こす：
// ::に続く名前は、クラス名か名前空間名でなければならない
typedef pointed_type<char*>::type result;
```

明らかに、`typename T::element_type`という式は、`T == char*`の場合、有効ではない。コンパイラはそれを指摘しているのである。選択コードの実装を手書きすれば、この問題を解決することが出来る：

```
namespace aux {
// 一般的な場合
template< typename T, bool is_pointer = false >
struct select_pointed_type
{
    typedef typename T::element_type type;
};

// 生ポインタのための特殊化
template< typename T >
struct select_pointed_type<T, true>
{
    typedef typename boost::remove_pointer<T>::type type;
};
}
```

¹ Tがポインタの場合を見分けるために、`pointed_type`を部分特殊化で実装するのは簡単だが、`if_`はもっと複雑な条件を扱うには適したツールである。説明のためであり、どうか疑わないでほしい。

```

};
}

template< typename T >
struct pointed_type
    : aux::select_pointed_type<
        T, boost::is_pointer<T>::value
    >
{
};

```

しかしこれは、もし繰り返し行うことが必要ならすぐにひどいことになる。そして、部分特殊化が使えない場合にはもっと悪化する。次のように、この問題を扱えるようにすることが出来る:

```

namespace aux {
template< typename T >
struct element_type
{
    typedef typename T::element_type type;
};
}

template< typename T >
struct pointed_type
{
    typedef typename mpl::if_<
        boost::is_pointer<T>
        , typename boost::remove_pointer<T>::type
        , typename aux::element_type<T>::type
    >::type type;
};

```

しかし、これもまたうまくいかない。aux::element_type<T>のネストした typeメンバへのアクセスにより、コンパイラが element_type<T>を T == char* でインスタンス化しなければならないなら、このインスタンス化は当然、無効である。また、先ほどの場合ならばコンパイルエラーにはならないが、boost::remove_pointer<T>テンプレートは常に同じように、同じ理由(ネストしたtypeメンバ)でインスタンス化される。テンプレートの「重さ」(どれほどのインスタンス化がコンパイラに課せられるか)に依存するが、致命的でない、不要なインスタンス化は問題になるかもしれないし、ならないかもしれない。しかし、一般的な大雑把なやり方で、そのようなコードを避けることが出来る。

先ほどのエラーに戻れば、上のコードをコンパイルするために、if_の呼び出しの外で、aux::element_type<T>に対し、そのネストしたtypeを「問い合わせる」振る舞いという要素が必要である。boost::remove_pointer<T>特性テンプレートと、aux::element_type<T>の両方が、結果型のために、習慣化された同じ名前を使っている、という事実により、リファクタリングはより簡単になる:

```

template< typename T >
struct pointed_type
{
private:
    typedef typename mpl::if_<
        boost::is_pointer<T>
        , boost::remove_pointer<T>
        , aux::element_type<T>
    >::type func_;

public:
    typedef typename func_::type type;
};

```

これで、コンパイラがboost::remove_pointer<T>、aux::element_type<T>の両方のインスタンス化

が起こらないことが保証される。たとえ、`if_`テンプレートのパラメータとして実際に使われてでも、である。これにより、`func_`として最終的に選ばれない限り、`aux::element_type<char*>`から逃れることが出来る。

上の技はテンプレートメタプログラムでは広く使われている。`if_`と高水準の等価性をもつものを導入することで、ネストした`type`メンバの選択を可能にするのである。呼び出しの一部として`func_::type`操作([nullary]メタ関数クラス適用と呼ばれる)を行うものである。MPLはそのようなテンプレートを提供し、`apply_if`と呼ばれている。これを使えば、上のコードを次のようにシンプルに書き直すことが出来る:

```
template< typename T >
struct pointed_type
{
    typedef typename mpl::apply_if<
        boost::is_pointer<T>
        , boost::remove_pointer<T>
        , aux::element_type<T>
    >::type type;
};
```

この技を完全にレビューするために、多少難しい例を考えてみよう。`boost::remove_pointer`特性テンプレート[TTL]の高水準ラッパー定義したいとする。これはポインタ修飾を条件付ではがすものである。これを、`remove_pointer_if`と呼ぼう:

```
template<
    typename Condition
    , typename T
>
struct remove_pointer_if
{
    typedef typename mpl::if_<
        Condition
        , typename boost::remove_pointer<T>::type
        , T
    >::type type;
};
```

上のコードは最初はどうも行くが、先に述べた問題に直面することになる。`boost::remove_pointer<T>`は、その結果が使われることがなくてもインスタンス化される。メタプログラミングの世界では、コンパイル時間は重要な資源であり [Abr01]、それは不要なテンプレートインスタンス化により浪費される。`if_`の両方の引数がnullaryメタ関数クラス適用の結果である時に、この問題をどう対処するかについては、ちょうど今見てきた。しかしこの例では、引数のひとつ(T)はただの単純な型であり、リファクタリングは不可能のように思われる。

この状況を脱する最も簡単な方法は、`if_`に、Tの代わりに本当のnullaryメタ関数を渡すことである。これは、呼び出しによりTを返すものである。MPLはそのために単純な方法を提供する。Tと`if_`の代わりに、`identity<T>`と`apply_if`を使えばいいのである。

```
template<
    typename Condition
    , typename T
>
struct remove_pointer_if
{
    typedef typename mpl::apply_if<
        Condition
        , boost::remove_pointer<T>
        , mpl::identity<T>
    >::type type;
};
```

これで、望みどおりのものを手にすることが出来る。

2.2. メタ関数

2.2.1. 単純な形式

C++では、パラメータ付きコンパイル時計算を可能にする基本的な言語構築は、クラステンプレート ([IS098], 14.5.1 [temp.class]節)である。単純なクラステンプレートは、メタ関数のために選択可能な最も単純なモデルである：実際のテンプレートパラメータとして型と/または非型の引数を受け取り、インスタンス化は新しい型を「返す」。例えば、次のコードはその引数から派生した型を作る：

```
template< typename T1, typename T2 >
struct derive : T1, T2
{
};
```

しかし、このモデルは制限が多すぎる：メタ関数の結果をクラス型に制限するだけでなく、与えられたクラステンプレートの実体化に制限するのである。全て呑めた関数呼び出しが、余計な水準のテンプレートネストを必要とするという事実については何も言っていないのに。特定のメタ関数に対しては、これでいいのかもしれないが、例えばintを「返す」ようなことができなくなってしまうモデルでは、明らかに十分汎用的とは言えない。この基本的な要求を満たすために、戻り値を提供するためのネストした型に頼らなければならない。

```
template< typename T1, typename T2 >
struct derive
{
    struct type : N1, N2 {};
};
```

```
// 馬鹿げた特殊化, しかしint を“返す”
template<>
struct derive<void, void>
{
    typedef int type;
};
```

Veldhuizen [Vel95a] は「コンパイル時関数」としてのこの形式のクラステンプレートについて最初に語った。そして、Czarneckiと Eisenecker [CE00] は同様の述語(彼らは、我々同様「メタ関数」という単純な言葉も使った)として「テンプレートメタ関数」を導入した。Czarneckiと Eiseneckerはまた、単純なメタ関数表現の限界を認識し、我々が項2.2.3で議論する形式を提案した。

2.2.2. 高次のメタ関数

構文的に単純だが、単純なテンプレートメタ関数形式はCxx;の他の部分と必ずしも最適にやりとりできるわけではない。特に、単純なメタ関数形式は、その定義を不必要に見苦しく、退屈にし、高次のメタ関数(他のメタ関数を操作するメタ関数)と共に機能するしかない。単純なメタ関数を他のテンプレートに渡すために、テンプレートテンプレートパラメータを使う必要がある。

```
// F(T1, F(T2, T3)) を返す
template<
    template<typename> class F
    , typename T1
    , typename T2
    , typename T3
>
struct apply_twice
{
    typedef typename F<
```

```

    T1
    , typename F<T2, T3>::type
  >::type type;
};

// T1, T2, T3から派生した型を返す新しいメタ関数
template<
  typename T1
  , typename T2
  , typename T3
  >
struct derive3
  : apply_twice<derive, T1, T2, T3>
{
};

```

これは異なるように見えるが、機能する。²しかし、メタ関数からメタ関数を「返す」時に初めて、これが壊れているということに気づくのである。

```

// G s.t. G(T1, T2, T3) == F(T1, F(T2, T3)) を返す.
template< template<typename> class F >
struct compose_self
{
  template<
    typename T1
    , typename T2
    , typename T3
    >
  struct type
    : apply_twice<F, T1, T2, T3>
  {
  };
};

```

第一の、そして最も明らかな問題は、`compose_self`の適用結果がそれ自体、型でなくテンプレートである、ということ、このため通常の方法で他のメタ関数に渡すことができない。更に些細な問題だが、しかし、「返される」メタ関数は実際には我々が望むものではない。`apply_twice`の様に振る舞うが、ひとつの重要な点で異なっている：それは同一性だ。C++型機構では、`compose_self<F>::template type<T, U, V>`は、`apply_twice<F, T, U, V>`と同義ではない。そして、メタ関数を比較するメタプログラムならば、この事実を発見するだろう。

C++ は型とクラステンプレートテンプレートパラメータとの間に厳格な線引きを行うので、単純なメタ関数に頼れば、メタ関数とメタデータとの間に「壁」が作られて、メタ関数は第2のクラス群の状態へと格下げされる。例えば、型シーケンスの紹介を思い出せば、メタ関数のconsリストをつくる方法はないのである。

```
typedef cons<derive, cons<derive3, nil> > derive_functions; // error!
```

consセルの再定義を考えるかもしれないので、先頭要素として`derive`を渡すことができる：

```

template <
  template< template<typename T, typename U> class F
  , typename Tail
  >
struct cons:

```

しかし、別の問題が出てくる：C++テンプレートは型引数という点で多相であるが、テンプレートテンプレートパラメータという点では多相ではない。テンプレートテンプレートパラメータのアリティ(パラメータの数)は厳格に強制されるので、`derive3`をconsリストに埋め込むことはまだできない。さらに、型とメタ関数の間の多相はサポートされていない(コンパイラはどちらかであることを期

² 実際は既に壊れている：`apply_twice`はメタ関数の概念に合致させない。なぜなら、最初のパラメータとして、(型というより)テンプレートを要求するからである。これはメタ関数プロトコルを破壊する。

待する), そして, 既に見たように, 「返される」メタ関数の構文と意味は返される型のそれとは異なるのである. 単純なテンプレートメタ関数形式で全てをやろうとすれば, 高次のメタ関数の応用性をひどく制限し, また実装上の明白性, 単純性, ライブラリの大きさといった, あらゆるところに悪い効果を起こすことになる.

2.2.3. メタ関数クラス

幸運にも, 「ソフトウェアには別の水準の間接化によって解決できない問題はない」という自明の理がここで使える. メタ関数を一級オブジェクトの状態に昇進させるために, MPLは「メタ関数クラス」の概念を導入する.

```
// deriveのメタ関数クラス形式
struct derive
{
    template< typename N1, typename N2 >
    struct apply
    {
        struct type : N1, N2 {};
    };
};
```

この形式は, STLの関数オブジェクトとして知られているので, 解りやすいだろう. ネストした `apply` テンプレートは実行時の関数呼び出し操作と同じ役割を持つ. 実際, コンパイル時メタ関数クラスとメタ関数の関係は, 実行時関数オブジェクトと関数の関係と同じである.

```
// addの関数形式
template< typename T > T add(T x, T y) { return x + y; }
```

```
// addの関数オブジェクト形式
struct add
{
    template< typename T >
    T operator()(T x, T y) { return x + y; }
};
```

2.2.4. ひとつで十分?

メタ関数クラス形式は前に述べた通常の添付レースメタ関数の全ての問題を解決する: これは通常のクラスなので, コンパイル時メタデータシーケンスに置くことも可能だし, 別の関数から, 別のメタデータと同じプロトコルで使うことも可能だ. だから, 通常のメタデータとメタ関数に対して, それぞれ個別のアリティをサポートして演算を行う ライブラリコンポーネントを別々に提供するようなコードの重複は必要なくなる.

しかし, メタ関数クラスを, コンパイル時関数実体の代理として考えることは, 同様にコード重複の危険にさらされるだろう: もしライブラリ自身のプリミティブ, アルゴリズム, その他諸々が クラステンプレートとして表されているなら, より高次の関数のコンテキストではこれらのアルゴリズムを再利用できないか, 或いは, 全てのアルゴリズムを第2の形式で書き直さなければならない. 例えば, 2つの異なる `find` がある:

```
// ユーザフレンドリな形式
template<
    typename Sequence
    , typename T
>
struct find
{
    typedef /* ... */ type;
};

// “メタ関数クラス”形式
struct find_func
```

```
{
    template< typename Sequence, typename T >
    struct apply
    {
        typedef /* ... */ type;
    };
};
```

勿論、3番目の選択肢は、「ユーザフレンドリ」形式を完全になくして、次のように書くことである:

```
typedef mpl::find::apply<list, long>::type iter;
// 或いはもしこちらの方が好きならば,
// typedef mpl::apply< mpl::find, list, long >::type iter;
```

次のように書く代わりに:

```
typedef mpl::find<list, long>::type iter;
```

このユーザビリティはひどすぎる。ライブラリのアルゴリズムを直接呼び出すことは、別のアルゴリズムやメタ関数に引数としてアルゴリズムを渡すことよりもずっと多いからだ。

2.2.5. メタ関数からメタ関数クラスへ

このジレンマに対するMPLのとった答えはラムダ式である。ラムダはライブラリがカーリー化メタ関数を使いそれらをメタ関数クラスに変換することを可能にする仕組みである。これにより、findアルゴリズムをより告示のメタ関数に引数として渡したければ、次のように書けばよいのだ:

```
using namespace mpl::placeholder;
typedef mpl::apply< my_f, mpl::find<_1, _2> >::type result;
```

_1と_2はメタ関数クラスを生成する第1, 第2引数のプレースホルダである。これは下のように、ユーザがfindをコードで直接使いたい時に直感的な構文を維持する:

```
typedef mpl::find<list, long>::type iter;
```

ラムダの能力は項3でより詳細に述べる。

2.3. シーケンス, アルゴリズム, イテレータ

2.3.1. イントロダクション

(型)シーケンスに対するコンパイル時反復は、テンプレートメタプログラミングの基本的概念のひとつである。演算されるオブジェクト型の違いは、同じだが同一ではないコード/設計の可変性の最も共通する点である。そしてそのような設計が、メタプログラミングの直接の狙いになることもある。テンプレートははじめは、この実際的な問題(例えばstd::vector)を解決するために設計された。しかし、(スタンドアロンの型ではなく)型シーケンスに対する操作、反復にとって、既に定義された抽象化、構築なしに、また、現在の言語の能力を使ってこれらの構築をエミュレートする既知の技術を使わずには、高次のメタプログラミングを手助けする効果は、制限されるだろう。

Czarnecki と Eisenecker [CE98], [CE00] が最初に、コンパイル時型シーケンスとそれらにたいするいくつかのアルゴリズムを導入した。木、リストなどの共通データ構造をクラステンプレートの合成を利用してコンパイル時に表現するアイデアは、既に存在していたが(例えば、式テンプレートライブラリの多くは、式の"パース"プロセスの一部として、そのような木を構築している[Ve195b])。Alexandrescu [Ale01] は型リストとそれらに対するいくつかのアルゴリズムを、多くのデザインパターンを実装するために利用した; その時のコードはLokiライブラリとして知られている[Loki]

2.3.2. アルゴリズムとシーケンス

Boost Metaprogramming Libraryの多くのアルゴリズムはシーケンスを操作する。例えば、リストから型を検索するのは次のような感じだ:

```
typedef mpl::list<char, short, int, long, float, double> types;
typedef mpl::find<types, long>::type iter;
```

ここでは、`find`は2つのパラメータを受け取る - 検索する(型)シーケンスと、探し出したい型(`long`) - そして、イテレータ `iter`を返す。これは、シーケンスのうち、`iter::type` が`long`と同一である最初の要素を指している。もしそのような要素が存在しなければ、`iter`は`end<types>::type`と同一である。基本的に、これは、`std::list`や`std::vector`の中からある値を検索する方法と同じだが、`mpl::find`がシーケンスをひとつのパラメータとして受け取る点が異なる。`std::find`は2つのイテレータを受け取るからだ。他はほとんど同じだろう - 名前も同じだし、セマンティクスはかなり近い、イテレータがあり、また、加賀だけでなく、述語を使っても探すことができる。

```
typedef mpl::find_if< types, boost::is_float<_> >::type iter;
```

STLとの概念的/構文的類似は偶然ではない。STLの概念的枠組みをコンパイル時世界に再利用することで、シーケンスデータ構造を扱う際に、よく知った、健全なアプローチを利用できるようになる。プログラマがSTLによって既に知っているアルゴリズムとイディオムはコンパイル時にも再利用できる。これは、MPLの大きな強みのひとつだと思う。テンプレートメタプログラミングライブラリを構築しようとした、初期のものとは異なるのだ。

2.3.3. シーケンスコンセプト

上の`find`の例では、`mpl::list`テンプレートを使って作られたシーケンスの中から型を探した; しかしライブラリが提供するシーケンスは`list`だけではない。`mpl::find`も、ハードコーディングされた他のアルゴリズムでも、`list`シーケンスだけしか扱えないわけではない。`list`はMPLの順シーケンスコンセプトのひとつのモデルに過ぎない。そして`find`はこのコンセプトの要求を満たすものならなんでも大丈夫だ。MPLにおけるシーケンスコンセプトの階層構造は極めてシンプルである - シーケンスは`begin<>`と `end<>`によって、その要素の範囲へのイテレータが生成されるコンパイル時のあらゆる実体である; 順シーケンスはそのイテレータが順イテレータの要求を満たすシーケンスである; 双方向シーケンスはそのイテレータが双方向イテレータの要求を満たす順シーケンスである; 最後に、ランダムアクセスシーケンスは、そのイテレータがランダムアクセスイテレータの要求を満たす双方向シーケンスである。³

特定のシーケンスの実装から(イテレータによって)アルゴリズムを切り離すことで、メタプログラマは自分自身のシーケンス型を作成し、ライブラリの他の部分を自由に使うことができる。例えば、次のように、3つの型のシーケンスを扱う`tiny_list`を定義することができる:

```
template< typename TinyList, long Pos >
struct tiny_list_item;

template< typename TinyList, long Pos >
struct tiny_list_iterator
{
    typedef typename tiny_list_item<TinyList, Pos>::type type;
    typedef tiny_list_iterator<TinyList, Pos-1> prior;
    typedef tiny_list_iterator<TinyList, Pos+1> next;
};

template< typename T0, typename T1, typename T2 >
struct tiny_list
{
    typedef tiny_list_iterator<tiny_list, 0> begin;
```

³ これらのコンセプトのより正確な定義はライブラリリファレンスドキュメント[MPLR]にある。

```

typedef tiny_list_iterator<tiny_list, 3> end;
typedef T0 type0;
typedef T1 type1;
typedef T2 type2;
};

```

```

template< typename TinyList >
struct tiny_list_item<TinyList, 0>
{
    typedef typename TinyList::type0 type;
};

```

```

template< typename TinyList >
struct tiny_list_item<TinyList, 1>
{
    typedef typename TinyList::type1 type;
};

```

```

template< typename TinyList >
struct tiny_list_item<TinyList, 2>
{
    typedef typename TinyList::type2 type;
};

```

これは、mpl::list同様にライブラリのアルゴリズムで使うことができる:

```

typedef tiny_list< char, short, int > types;
typedef mpl::transform<
    types
    , boost::add_pointer<_1>
    >::type pointers;

```

tiny_listは双方向シーケンスのモデルである。 advanceとdistanceメンバをtiny_list_iteratorに追加すれば、 ランダムアクセスシーケンスになる:

```

template< typename TinyList, long Pos >
struct tiny_list_iterator
{
    static long const position = Pos;

    typedef typename tiny_list_item<TinyList, Pos>::type type;
    typedef tiny_list_iterator<TinyList, Pos-1> prior;
    typedef tiny_list_iterator<TinyList, Pos+1> next;

    template< typename N > struct advance
    {
        typedef tiny_list_iterator<
            TinyList
            , Pos + N::value
            > type;
    };

    template< typename Other > struct distance
    {
        typedef mpl::integral_c<
            long
            , Other::position - position
            > type;
    };
};

```

tiny_list自身は興味の対象ではない(結局それは3つの要素を保持できる)が、 もし上のテクニックが自動化され、 (5, 10, 20, ... の要素の)それほど小さくないシーケンスを定義できるなら、 価値ある

ことだろう。⁴

外部コード生成はひとつの選択肢だが、言語内での解決も存在する。しかしそれはテンプレートメタプログラミングではなく、プリプロセッサメタプログラミングである。実際、MPLのvector - ランダムアクセスイテレータを提供する固定サイズ型シーケンス - は上のtiny_listによく似た実装が行われている。そこでは、Boost Preprocessor ライブラリ [PRE]が使われている。

2.3.4. アドホックな例, 再び

このように、ライブラリはユーザに、STLフレームワークとほとんど完全に対応するコンパイル時フレームワークを提供する。これは、メタプログラミングのタスクを解決するのに役立つのだろうか? 以前の最大の例に戻って、MPLが提供するものを利用したよりよいやり方で書き直せるか見てみよう。実際、見るべきものはそう多くない、なぜならMPLの実装はワンライナーだから(それをここでは再適応性のために展開してただけのことだ)⁵ :

```
template< typename Sequence >
struct largest
{
    typedef typename mpl::max_element<
        Sequence
        mpl::less<
            mpl::sizeof_<_1>
            , mpl::sizeof_<_2>
        >
        >::type iter;

    typedef typename iter::type type;
};
```

トリッキーなパターンマッチングによる終了条件はもうないし、部分特殊化ももうない;そして、より重要なことだが、上のコードが、オリジナルのものについては誰も言えなかった何かしていることは明らかなのだ - 例え、全てがテンプレートであっても。

2.3.5. 主要反復アルゴリズムとしてのiter_fold

ライブラリの内部構造をもう少し調べるために、上の例でのmax_elementがどう実装されているのか見てみよう。今度こそはあのひどい部分特殊化、深遠なパターンマッチングなどが全部現れると思われるかもしれない。では、見てみよう:

```
template<
    typename Sequence
    , typename Predicate
>
struct max_element
{
    typedef typename mpl::iter_fold<
```

⁴ ランダムアクセスはコンパイル時においても、実行時同様に重要である。例えば、ソート済みランダムアクセスシーケンスから、lower_boundを使って検索することは、同じ操作を順アクセスしかできないlistに行くより遙かに速い。

⁵ これは別の、よりエレガントな実装である:

```
template< typename Sequence >
struct largest
{
    typedef typename mpl::max_element<
        mpl::transform_view<
            Sequence
            , mpl::sizeof_<_>
        >
        >::type type;
};
```

```

Sequence
, typename mpl::begin<Sequence>::type
, if_< less< deref<_1>, deref<_2> >, _2, _1 >
>::type type;
};

```

ここで注意すべきことはまず、このアルゴリズムは別のものを実装されている、ということだ。つまりそれは、`iter_fold`。実際、これはおそらく、この例の最も重要な点である。なぜなら、ライブラリにあるほとんど全ての汎用シーケンスアルゴリズムは、`iter_fold`を使って実装されているからだ。もしユーザが自分用のシーケンスアルゴリズムを必要とするなら、まず間違いなくこのプリミティブを使うことができる。これは、頑張って反復を作ったり、ループ終了のために特殊な場合のパターンマッチングをしたり、部分特殊化がないために代替手段を作ったり、といったことが必要ない、ということの意味する。そしてまた、そのアルゴリズムは自動的にライブラリが実装した最適化の恩恵を受けることができるし（例えば再帰の展開）、順シーケンスのモデルならばどんなシーケンスでも扱える、ということも意味する。なぜなら`iter_fold`はそのシーケンス以上なにも要求しないからだ。

`iter_fold`アルゴリズムは基本的に、多くの関数的プログラミング言語で基本的な、よく知られたプリミティブを構成する`fold`や`reduce`関数の、コンパイル時の対応物である。C++プログラマにとってより解りやすいアナロジーはC++標準ライブラリ([IS098], 26.4.1項 [lib.accumulate])の`std::accumulate`アルゴリズムである。しかし、`iter_fold`は再帰巡回の自然な性質の利益を受けるように設計されている：これは、2つのメタ関数クラス引数を受け取り、その最初のものは“入りの”状態に適用され、二番目のものは“出の”状態に適用される。

MPLで定義される`iter_fold`のインタフェースは以下の通り：

```

template<
    typename Sequence
    , typename InitialState
    , typename ForwardOp
    , typename BackwardOp = _1
>
struct iter_fold
{
    typedef /*unspecified*/ type;
};

```

アルゴリズムは2項`ForwardOp`と`BackwardOp`操作の2方向の連続した適用結果を「返す」。これらの操作は[`begin<Sequence>::type`, `end<Sequence>::type`]の範囲のイテレータと、演算の前の結果に対して行われる；`InitialState`は論理的にシーケンスの前に置かれ、順巡回に含まれる。結果型は、もしシーケンスがemptyなら、`InitialState`と同一である。

ライブラリはまた、`iter_fold_backward`, `fold`, `fold_backward`アルゴリズムを提供する。これらは`iter_fold`のラップであり、よく使われるパターンのために存在する。

2.3.6. 数値シーケンス

今まで見てきたものは、型シーケンス(及び型シーケンスのアルゴリズム)である。同じように、ライブラリを使ってコンパイル時値を操作することは可能だし簡単だ。覚えておくべき唯一のことは、C++では、クラステンプレートの非型テンプレートパラメータは非多相的な振る舞いのもう一つの例である、ということだ。言い換えれば、もしメタ関数を非型テンプレートパラメータ(例えば`long`)を受け取るように定義したら、コンパイル時汎整数定数以外の何も、それには渡すことはできない。

```

template< long N1, long N2 >
struct equal_to
{
    static bool const value = (N1 == N2);
};

```

```
equal_to<5, 5>::value; // ok
equal_to<int, int>::value; // error!
```

もちろん次のものも機能しない

```
typedef mpl::list<1, 2, 3, 4, 5> numbers; // error!
```

これは明らかな制限だが、ライブラリの設計にとって別のジレンマを課してくる：一方で、ユーザに対して型操作だけに制限したくない。他方、整数演算を完全にサポートするには、少なくともほとんどのライブラリの機能を二度書きしなければならない。6 - メタ関数を通常のクラステンプレートで表現するかどうか、という状況とおなじである。この問題に対する解決は、同様である：整数値を型でラップするのだ7。例えば、数値リストを作るのには次のように書ける：

```
typedef mpl::list<
    mpl::int_c<1>
    , mpl::int_c<2>
    , mpl::int_c<3>
    , mpl::int_c<4>
    , mpl::int_c<5>
> numbers;
```

汎整数定数値を型にラップすることで第一クラス群にすることは、メタプログラミングの内側では非常に重要である。しかし、もし使っているメタ関数が、型、整数値、他のメタ関数、或いは他の何か、例えば固定小数点、有理数(mpl::fixed_cとmpl::rational_c)に対する演算だとしても、多くの場合それを知らない(そして気付きもしない)。

しかし、ユーザにとってみれば、上の例は短くて不当なものより遙かに冗長だ。このように、利便性のために、ライブラリはユーザに非型テンプレートパラメータを受け取るテンプレートを与えているが、より簡潔な記述も提供している：

```
typedef mpl::list_c<long, 1, 2, 3, 4, 5> numbers;
```

同様に、vectorもある：

```
typedef mpl::vector_c<long, 1, 2, 3, 4, 5> numbers;
```

2.3.7. いろいろなシーケンス

C++に汎用的なメタプログラミング機構を提供するというこれまでの努力は、常に **コンス**-スタイルの型リストと、size, atの様な、特定のシーケンス実装と結びついた、いくつかのコアとなるアルゴリズムに向けられてきた。そのようなシステムは、純粋関数指向Lispの機能性を思い起こさせるエレガントな単純性を持っている。実行時ライブラリ(特にSTL)に対応するものによって提供された、シーケンスアルゴリズムの基本セットでさえ、実装には大変時間がかかったが、もしSTLから何も学んでいなかったら、これらのアルゴリズムの実装を特定のシーケンスの実装に結びつけることは、間違った努力だったに違いない！

真実は、ひとつの「最良の」型シーケンスの実装などない、ということだ。その理由は、ひとつの「最良の」実行時シーケンスの実装が決して存在しないことと同じである。さらに、すでにかなりの数の型リスト実装が現在使われている；また、STLアルゴリズムがSTLコンテナ以外のシーケンスを操作できるように、MPLアルゴリズムも異なる型シーケンスで機能するように設計されている。

型リストだけが役に立つコンパイル時シーケンスではない、という事実に驚く人もいるだろう。再び、C++標準ライブラリにリスト、ベクタ、デキュー、セットなどがあるのと同じ理由で、様々なコンパイル時コンテナが必要なのだ - 異なるコンテナは異なる機能と動作特性を持ち、特定のアルゴリズムでの応用可能性や効率だけでなく、それらを使うコードの表現性や冗長性も決定する。実行

6 理想的には、もしこの方法をとれば、全てのテンプレートは全ての整数型 -char, int, short, longなど- に対してそれぞれ実装しなければならない。

7 同じテクニックが Czarnecki と Eisenecker によって[CE00]で提案されている。

時のパフォーマンスはC++メタプログラミングにとって問題ではないが、コンパイル時間は最先端のC++ソフトウェア開発では重要なボトルネックになる[Abr01].

MPLは5種類の組み込みシーケンスを提供している: `list`, `list_c` (これは値ラップの`list`), 最大サイズ固定のランダムアクセスシーケンスである`vector`, `vector_c`, そして, 連続する整数値のランダムアクセスシーケンスである`range_c`である. しかし, より重要なことは, 任意のシーケンス型に適応する能力である. シーケンスがライブラリアルゴリズムで使われるために提供しなければならない唯一のコアの操作は, `begin<>`と`end<>`メタ関数である. これは, シーケンスのイテレータを”返す”. STLの様に, ライブラリが提供する汎目的シーケンスアルゴリズムのほとんどを実装するために使われるものは, イテレータである. 同じくSTLの様に, アルゴリズムの特殊化が特定のシーケンスについての実装の知識を利用するために使われる: `back<>`, `front<>`, `size<>` `at<>`の様な, ”基本”シーケンス操作はシーケンス型に対し特殊化することで, 完全な汎用版よりも効率的な実装を提供している.

2.3.8. ループ/再帰の展開

ほとんど偶然の一致だが, ループ展開はコンパイル時反復アルゴリズムにとって, 実行時アルゴリズム同様重要である. その理由を知るには, C++メタプログラミングでの全ての”ループ”が, 実際, 再帰によって実装されていて, テンプレートインスタンス化深度はコンパイラの実装の価値ある資源なのだ, ということ覚えておかなければならない. 事実, C++標準 Annex B([IS098], annex B [limits])では, 最小でも17回の再帰的にネストしたテンプレートインスタンス化を推奨している; しかし, 気合いの入った多くのメタプログラムにとって, これではあまりにも少なすぎる. そのいくつかでは, 優れたコンパイラの, 頑張っているインスタンス化制限をも簡単に突破してしまう. これがどう振る舞うかを見るために, アルゴリズムの状態とシーケンスのそれぞれの要素を組み合わせる `fold`メタ関数の単純な実装を調べてみよう.

```
namespace aux {

// 非特殊化版は内部状態と最初の要素を組み合わせ, 残りの処理を再帰呼び出しする
template<
    typename Start
    , typename Finish
    , typename State
    , typename BinaryFunction
>
struct fold_impl
: fold_impl<
    typename Start::next
    , Finish
    , typename apply<
        BinaryFunction
        , State
        , typename Start::type
    >::type
    , BinaryFunction
>
{
};

// ループ終了のための特殊化版
template<
    typename Finish
    , typename State
    , typename BinaryFunction
>
struct fold_impl<Finish, Finish, State, BinaryFunction>
{
    typedef State type;
};
```

```

} // namespace aux

// public interface
template<
    typename Sequence
    , typename State
    , typename ForwardOp
>
struct fold
    : aux::fold_impl<
        , typename begin<Sequence>::type
        , typename end<Sequence>::type
        , State
        , typename lambda<ForwardOp>::type
    >
{
};

```

シンプルでエレガントだけど、この実装は常に、少なくとも入力シーケンスの数と同じレベルの再帰的テンプレートインスタンス化を必要とする。⁸ ライブラリは再帰の明示的“展開”によって、この問題を対処している。foldの例にこのテクニックを使うために、アルゴリズムの1段階を洗い直すことから始める。fold_impl_stepメタ関数は2つの結果を持つ: type (次の状態)と, iterator (次のシーケンス位置)である。

```

template<
    typename BinaryFunction
    , typename State
    , typename Start
    , typename Finish
>
struct fold_impl_step
{
    typedef typename apply<
        BinaryFunction
        , State
        , typename Start::type
    >::type type;

    typedef typename Start::next iterator;
};

```

主要なアルゴリズムの実装のように、ループ終了条件を特殊化しているので、そのステップはno-opになる。

```

template<
    typename BinaryFunction
    , typename State
    , typename Finish
>
struct fold_impl_step<BinaryFunction, State, Finish, Finish>
{
    typedef State type;
    typedef Finish iterator;
};

```

ここでは、単純にN回のfold_impl_stepの呼び出しを挿入することで、あらゆる定数要因Nによるfoldのインスタンス化深度を減らしている。

```

template<
    typename Start
    , typename Finish

```

⁸ apply<...>式の複雑さによっては、それ以上になる。全ての再帰深度に対して、この深度が加わるのである。

```

    , typename State
    , typename BinaryFunction
    >
struct fold_impl
{
private:
    typedef fold_impl_step<
        BinaryFunction
        , State
        , Start
        , Finish
    > next1;

    typedef fold_impl_step<
        BinaryFunction
        , typename next1::type
        , typename next1::iterator
        , Finish
    > next2;

    typedef fold_impl_step<
        BinaryFunction
        , typename next2::type
        , typename next2::iterator
        , Finish
    > next3;

    typedef fold_impl_step<
        BinaryFunction
        , typename next3::type
        , typename next3::iterator
        , Finish
    > next4;

    typedef fold_impl_step<
        typename next4::iterator
        , Finish
        , typename next4::type
        , BinaryFunction
    > recursion;

public:
    typedef typename recursion::type type;
};

```

MPLでは、展開の要因を、使われているC++の実装の要求に合わせて、また、ユーザが値を書き換えることが可能な選択しと共に、この展開テクニックを全てのアルゴリズムで使っている。⁹ この事実により、ユーザは、通常はもっと単純なアルゴリズムの実装に当たり出会うような、メタプログラミングの限界を超えることができる。経験によれば、いくつかのコンパイラではループ展開を使えば、メタプログラムインスタンス化の速度は僅かだが遅くなった(多くても10%程度だろう)。

3. ラムダ機構

MPLのラムダ機構により、クラステンプレートを「ラムダ式」にインライン合成することが可能になる。ラムダ式はクラスであり、通常メタ関数クラスのように渡すことが可能であり、またその式

⁹ この実装の詳細は比較的楽にできた。これは、Boost Preprocessor ライブラリによるところが大きい。保守するにはコードを一度コピーするだけでよかったのだ。

を使った適用の前にメタ関数クラスに変形することが可能である.

```
typedef mpl::lambda<expr>::type func;
```

例えば, Boost `type_traits` ライブラリ [TTL] の `boost::remove_const` 特性テンプレートは, (明らかに) クラステンプレートであり, MPL の用語ではメタ関数である. その「インライン合成」の最も単純な例は次のようなものだろう:

```
typedef boost::remove_const<_1> expr;
```

この形式は「ラムダ式」と呼ばれる.

```
typedef boost::remove_const<_1> expr;
typedef mpl::lambda<expr>::type func;
```

`func` は単項メタ関数クラスであり, そのような使い方ができる. 特に, 渡すことができるし, 呼び出す(適用する)ことができる.

```
typedef mpl::apply<func, int const>::type res;
BOOST_MPL_ASSERT_IS_SAME(res, int);
```

または,

```
typedef func::apply<int const>::type res;
BOOST_MPL_ASSERT_IS_SAME(res, int);
```

インライン合成は, 明白な式を作るので, メタ関数を扱う時に構文的に非常に力強い:

```
typedef mpl::logical_or<
    mpl::less< mpl::sizeof<_1>, mpl::int_c<16> >
    , boost::is_same<_1, _2>
> expr;
```

```
typedef mpl::lambda<expr>::type func;
```

最後の部分 (`typedef lambda<expr>::type func`) を明示する必要はない. なぜなら全てのアルゴリズムは, 内部的にそれらのメタ関数クラスオペランドに対してこれを行っているからである (メタ関数に適用される `lambda<T>::type` 式は同じメタ関数クラスに返されるので, 無条件に式を適用することは安全である).

上のメタ関数クラスと同じようなものを書くもう一つの方法は, 次の通り:

```
typedef bind<
    mpl::meta_fun2<mpl::logical_or>
    , mpl::bind< mpl::meta_fun2<mpl::less>
        , mpl::bind< mpl::meta_fun1<mpl::sizeof<_>, _1 >
        , mpl::int_c<16>
    >
    , mpl::bind< mpl::meta_fun2<boost::is_same>, _1, _2 >
> func;
```

同じ方法で, `mpl::compose_` の仲間のテンプレートを使うこともできる. ここではメタ関数をメタ関数クラスに変換するために `mpl::meta_fun` テンプレートを使い, それから `mpl::bind` を使ってそれらを組み合わせている. ここ形式から上のインラインラムダ式への変形, そしてその逆の変形は, 機構的なものであり, それは基本的に, `typedef mpl::lambda<expr>::type` 式が行っていることである.

独自のメタ関数(アルゴリズム, プリミティブ, など)の為に, MPL は上のことをもっと簡単に書けるようにしている:

```
typedef mpl::bind<
    mpl::logical_or<>
    , mpl::bind< mpl::less<>, mpl::bind<mpl::sizeof<_>, _1>, mpl::int_c<16> >
>
```

```

    , mpl::bind< mpl::make_f2<boost::is_same>, _1, _2 >
    > func;

```

ここではまだ、`is_same`を`make_f2`でラップする必要がある。なぜならそれは、(MPL)が知らないテンプレートだから。

ここで、クラステンプレートメタ関数とメタ関数クラスをひとつのラムダ式で組み合わせることについて、次のように行うことが可能である：

```

struct my_predicate
{
    template< typename T1, typename T2 > struct apply
    {
        //...
    };
};

typedef mpl::logical_or<
    mpl::less< mpl::sizeof_<_>, mpl::int_c<16> >
    , mpl::bind< my_predicate, _, _ > // here
    > expr;

```

何かをその引数のひとつに結びつける(或いはパラメータの順番を変更する)には、次のどちらかを使えばよい：

```

typedef mpl::logical_or<
    mpl::less< mpl::sizeof_<_>, mpl::int_c<16> >
    , mpl::bind<my_predicate, int, _>::type // here
    > expr;

```

または

```

typedef mpl::logical_or<
    mpl::less< mpl::sizeof_<_>, mpl::int_c<16> >
    , my_predicate::apply<int, _> // here
    > expr;

```

4. コード生成機構

特に数値演算の領域では、計算のいくつかの部分をコンパイル時に行いたい、結果をさらに処理するためにプログラムの実行時部分に渡したい時があるだろう。例えば、固定小数点演算を行う複雑なコンパイル時アルゴリズムを実装したと仮定しよう：

```

// 固定小数点アルゴリズム入力
typedef mpl::vector<
    mpl::fixed_c<-1, 2345678>
    , mpl::fixed_c<9, 0001>
    // ..
    , mpl::fixed_c<3, 14159>
    > input_data;

/*
  複雑なコンパイル時アルゴリズム
*/
typedef /*...*/ result_data;

```

ここでの`result_data`が、アルゴリズムの結果を保持する`mpl::fixed_c`型のシーケンスであり、結果を実行時のアルゴリズムに渡したいとしよう。MPLでは、これは次のようにできる：

```

double my_algorithm()
{

```

```

// プログラムの実行時部分に結果を渡す
std::vector<double> results;
results.reserve(mpl::size<result_data>::value);
mpl::for_each<numbers, _>(
    boost::bind(&std::vector<double>::push_back, &results, _1)
);
// ...
}

```

`for_each<numbers, _>(...)`の呼び出しは、コンパイル時の`result_data`を 実行時の`results`に実際に移すものである。 `for_each`は次のように宣言された関数テンプレートだ:

```

template<
    typename Seq
    , typename TransformOp
    , typename F
>
void for_each(F f)
{
    // ...
}

```

関数を呼び出すために、2つの実際のテンプレートパラメータを明示的に与えなければならない。 コンパイル時シーケンス`Seq`と、単項変形関数`TransformOp`、 それに実行時関数の引数`f`(この例では`numbers, _`、 `boost::bind(...)`がそれぞれ対応する)である。 `f`は関数オブジェクトであり、その`operator()`は、`TransformOp`により変形された`Seq`の全ての要素に対して呼び出される。

これを上の例に使えば:

```

mpl::for_each<numbers, _>(
    boost::bind(&std::vector<double>::push_back, &results, _1)
);

```

これは、次のことと大体同じ関数呼び出しを行う:

```

f(mpl::apply< _, mpl::at_c<result_data, 0>::type >::type());
f(mpl::apply< _, mpl::at_c<result_data, 1>::type >::type());
// ...
f(mpl::apply< _, mpl::at_c<result_data, n>::type >::type());

```

但し、 `n == mpl::size<result_data>::type::value` である。

5. 例: コンパイル時 FSM ジェネレータ

有限状態マシン (FSM) は、プログラムの振る舞いを記述し実装する重要な道具である [HU79], [Mar98]. また、コードの中にこれらの単純な数学的モデルを実装するために、行わなければならない繰り返しや、決まり文句的な演算を削減するために、metaprogrammingが利用できる領域の例として、ちょうどよい。以下、Boost Metaprogramming Library機構を使って実装した単純な状態マシンジェネレータを示す。ジェネレータはコンパイル時オートマトン記述を受け取り、それを実行時にFSMを実装するC++のコードに変換する。

FSMの記述は基本的に、状態と、それら全てを一緒に結びつける、イベントと状態遷移テーブル (STT) の組み合わせである。ジェネレータはテーブルを通して、FSMのエッセンスである、状態マシンのイベント処理を生成する。

有限状態マシンモデルをつかって単純なミュージックプレイヤーを実装したいとしよう。FSMの状態遷移テーブルは表 1に示してある。STTの形式は平文英語でFSMの振る舞いを記述する通常の方法に従っている。例えば、表の第1行は次のように読む: 「もしモデルが`stopped`状態で、`play_event`を受け取ったなら、`do_play`遷移関数が呼ばれ、モデルは`playing`状態に遷移する。」

状態	イベント	次の状態	遷移関数
stopped	play_event	playing	do_play
playing	stop_event	stopped	do_stop
playing	pause_event	paused	do_pause
paused	play_event	playing	do_resume
paused	stop_event	stopped	do_stop

表 1. プレイヤのアクション及び状態遷移テーブル

遷移表は対象となるFSMの完全な形式定義を与える。そして、その定義をコードにする方法は多くある。例えば、もし列挙型の数値として状態を定義し、基本となるeventクラスのいくつかから派生したクラスとしてイベントを定義したなら、¹⁰ 次のようになる:

```
class player
{
public:
    // イベント宣言
    struct event;
    struct play_event;
    struct stop_event;
    struct pause_event;

    // “入力”関数
    void process_event(event const&); // throws

private:
    // 状態
    enum state_t { stopped, playing, paused };

    // 遷移関数
    void do_play(play_event const&);
    void do_stop(stop_event const&);
    void do_pause(pause_event const&);
    void do_resume(play_event const&);

private:
    state_t m_state;
};
```

最も単純な方法で上の表からFSMの実装を派生させるなら、次のようになる:

```
void player::process_event(event const& e)
{
    if (m_state == stopped)
    {
        if (typeid(e) == typeid(play_event))
        {
            do_play(static_cast<play_event const&>(e));
            m_state = playing;
            return;
        }
    }
    else if (m_state == playing)
    {
        if (typeid(e) == typeid(stop_event))
        {
```

¹⁰ イベントはアクション関数に渡される必要がある、なぜならそれらはアクションのための、イベント特有の情報を持っているから。

```

        do_stop(static_cast<stop_event const&>(e));
        m_state = stopped;
        return;
    }

    if (typeid(e) == typeid(pause_event))
    {
        do_pause(static_cast<pause_event const&>(e));
        m_state = paused;
        return;
    }
}
else if (m_state == paused)
{
    if (typeid(e) == typeid(stop_event))
    {
        do_stop(static_cast<stop_event const&>(e));
        m_state = stopped;
        return;
    }

    if (typeid(e) == typeid(play_event))
    {
        do_play(static_cast<play_event const&>(e));
        m_state = playing;
        return;
    }
}
else
{
    throw logic_error(
        boost::format("unknown state: %d")
            % static_cast<int>(m_state)
    );
}

throw std::logic_error(
    "unexpected event: " + typeid(e).name()
);
}

```

ネストしたif(またはswitch-case)を使ってFSMの構造を実装することは特に間違いではないが、このアプローチの弱さは明らかに、上のコードの多くが、決まり文句的なものである、ということだ。決まり文句のコードでは何をするかという、それをコピペし、名前などを変えて、新しい場所に合わせて調節するだろう；そしてそこが、最もエラーが混ざり込みやすい場所なのだ。絵弁と処理の全ての行は(構造的に)同じように見えるのだから、変えられるべきことを忘れていても、それは容易に見過ごされ、多くのエラーは実行時まで現れないのだ。

上のFSMの遷移表は、たった5行の長さである；理想的には、オートマトンの統制ロジックのスケルトンの実装を、これくらい短くできればよい(あるいは、少なくとも、これくらい短く見ればよい、つまり、気にしなくていいようにある形式にカプセル化するのだ)。

5.1. 実装

C++のプログラムでSTTを表現するために、テーブルの一行を表現するtransitionクラステンプレートを実装する。テーブル自体は、そのような行のシーケンスとして表現される：

```

typedef mpl::list<
    transition<stopped, play_event, playing, &player::do_play>
    , transition<playing, stop_event, stopped, &player::do_stop>
    , transition<playing, pause_event, paused, &player::do_pause>

```

```

    , transition<paused, play_event, playing, &player::do_resume>
    , transition<paused, stop_event, stopped, &player::do_stop>
>::type transition_table;

```

ここで、完全なFSMは次のようになる:

```

class player
  : state_machine<player>
{
private:
    typedef player self_t;

    // 状態不変関数
    void stopped_state_invariant();
    void playing_state_invariant();
    void paused_state_invariant();

    // 状態(不変関数が非型テンプレートパラメータとして渡され、
    // 呼び出され、FSMは対応する状態になる。
    typedef state<0, &self_t::stopped_state_invariant> stopped;
    typedef state<1, &self_t::playing_state_invariant> playing;
    typedef state<2, &self_t::paused_state_invariant> paused;

private:
    // イベント宣言; イベントは型として表現され、
    // それぞれのイベントに固有のデータを運ぶ;
    // しかしジェネレータにとっては必要ではないので、後に定義する。
    struct play_event;
    struct stop_event;
    struct pause_event;

    // 遷移関数
    void do_play(play_event const&);
    void do_stop(stop_event const&);
    void do_pause(pause_event const&);
    void do_resume(play_event const&);

    // STT
    friend class state_machine<player>;
    typedef mpl::list<
        transition<stopped, play_event, playing, &player::do_play>
        , transition<playing, stop_event, stopped, &player::do_stop>
        , transition<playing, pause_event, paused, &player::do_pause>
        , transition<paused, play_event, playing, &player::do_resume>
        , transition<paused, stop_event, stopped, &player::do_stop>
    >::type transition_table;
};

```

これが全てだ - 上のコードは我々のしように従う完全なFSMの実装を生成する。 それを使う前に唯一必要なことは、(前に前方宣言された)イベント型の定義である:

```

// イベント定義
struct player::play_event
  : player::event
{
};

// ...

```

使い方も簡単:

```

int main()
{
    // usage example

```

```
player p;  
p.process_event(player::play_event());  
p.process_event(player::pause_event());  
p.process_event(player::play_event());  
p.process_event(player::stop_event());  
return 0;  
}
```

5.2. 関連する研究

C++での汎目的状態マシン実装の分野で、特筆すべき先行研究としては、Robert Martin の状態マシンコンパイラ [SMC]がある。SMCはマシンの状態遷移テーブルのASCII記述を受け取り、ある種の状態デザインパターン [Hun91], [GHJ+95] を利用して、FSMを実装するC++コードを生成する。Lafreniere [Laf00] は別のアプローチを提案している。それは、外部のツールを使わずに、FSMはテーブルドリブンである。

6. 謝辞

Peter Dimov はbindの機能に貢献してくれた。これなしには、コンパイル時ラムダ式は不可能である。MPLの実装は Vesa Karvonen の素晴らしいBoost Preprocessor ライブラリがなければ、全然違ったものになっていただろう。著者はまた、David B. Held に感謝している。このドキュメントを書く際に、ずっと親切に協力してくれた。もちろん、残っているエラーは我々の責任だ。

References

- [Abr01] David Abrahams , Carlos Pinto Coelho, Effects of Metaprogramming Style on Compilation Time, 2001
- [Ale01] Andrei Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley, ISBN 0-201-70431-5, 2001
- [CE98] Krzysztof Czarnecki , Ulrich Eisenecker, Metalisp, <http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm>
- [CE00] Krzysztof Czarnecki , Ulrich Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, ISBN 0-201-30977-7, 2000
- [EBNF] ISO/IEC 14977:1996(E), Information technology ▪ Syntactic metalanguage ▪ Extended BNF, ISO/IEC, 1996
- [GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, , John Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, ISBN 0-201-63361-2, 1995
- [HU79] Hopcroft , Ullman, Introduction to automata theory, languages and computations, Addison-Wesley, 1979
- [Hud89] Paul Hudak, Conception, Evolution, and Application of Functional Programming Languages, ACM Computing Surveys, Association for Computing Machinery (ACM), ISSN 0360-0300, Vol. 21, No. 3, pp. 359-411, September, 1989
- [Hun91] Immo Huneke, Finite State Machines: A Model of Behavior for C++, C++ Report, SIGS Publications Inc., ISSN 1040-6042, 1991
- [IS098] ISO/IEC 14882:1998(E), Programming languages ▪ C++, ISO/IEC, 1998
- [Joh79] Stephen C. Johnson, Yacc: Yet Another Compiler Compiler, UNIX Programmer's Manual, Vol. 2b, pp. 353-387, 1979
- [Laf00] David Lafreniere, State Machine Design in C++, C/C++ User Journal, CMP Media LCC, ISSN 1075-2838, Vol. 18, No. 5, May 1998
- [Loki] The Loki library, <http://sourceforge.net/projects/loki-lib/>
- [Mar98] Robert C. Martin, UML Tutorial: Finite State Machines, C++ Report, SIGS Publications Inc., ISSN 1040-6042, June 1998
- [MPLR] Boost MPL Library Reference Documentation, http://www.mywikinet.com/mpl/ref/Table_of_Content.html
- [PRE] Vesa Karvonen, Boost Preprocessor Metaprogramming library, <http://www.boost.org/libs/preprocessor/doc/>
- [SMC] Robert C. Martin, SMC - Finite State Machine Compiler (C++), <http://www.objectmentor.com/resources/downloads/index>
- [STL94] A. A. Stepanov , M. Lee, The Standard Template Library, Hewlett-Packard Laboratories, 1994
- [SPL] Boost Smart Pointer library, http://www.boost.org/libs/smart_ptr/
- [SS75] Gerald ▪ J. Sussman , Guy ▪ L. Steele Jr., Scheme: An interpreter for extended lambda calculus, MIT AI Memo 349, Massachusetts Institute of Technology, May 1975

[TTL] Boost Type Traits library, http://www.boost.org/libs/type_traits/

[Vel95a] Todd Veldhuizen, Using C++ template metaprograms, C++ Report, SIGS Publications Inc., ISSN 1040-6042, Vol. 7, No. 4, pp. 36-43, May 1995

[Vel95b] Todd Veldhuizen, Expression templates, C++ Report, SIGS Publications Inc., ISSN 1040-6042, Vol. 7, No. 5, pp. 26-31, Jun 1995

[Unr] Erwin Unruh, Prime number computation, ANSI X3J16-94-0075/ISO WG21-462