7J-07

C++における型制約用メタ関数の簡略化

滝　直也　　森山甲一　　武藤敦子　　犬塚信博
名古屋工業大学

1　はじめに

ジェネリックプログラミングにおいて，型引数を制約する型制約という概念がある．C++17 までの言語機能のみで型制約を実現するものとして真理値を取得するメタ関数が知られるが，その記述には多くのメタプログラミングの知識が必要であり，複雑なコードになりがちである．それを解決するものとして requires 式 [1] が提案されているが，まだ実装されていない．

本研究では，requires 式のアイデアを用いた関数として制約を記述することでメタ関数を簡略化する方法を提案する．いくつかの例で実装をして既存の手法と比較を行い，メタ関数の可読性の向上，及びコード量の削減を確認した．

2　C++のテンプレート

2.1　概要

ジェネリックプログラミングとは，データ型に依存しないプログラミング方式であり，C++ではテンプレートと呼ばれる機能を使用することでこれを実現できる．テンプレートは型引数としてデータ型を与えることで使用でき，コンパイラは仮引数を与えられた実引数に全て置換することでコードを実体化する．

2.2　短所

テンプレートの短所の一つとして，間違ったデータ型を与えた際にコンパイルが進んだ時点でしかエラーがわからず，結果として大量でわかりにくいエラー文が出ることがあげられる．利用者側はあらかじめ実引数への要求を把握していなければならない．

そのため，型引数に使用できる型を制限し，間違った型を使用した時点でエラーにする型制約が必要とされる．しかし，現時点の C++には言語機能としての型制約は存在しない．

3　従来のメタ関数の記述

真理値を取得するメタ関数を用意できれば既存の言語機能のみで型制約を実現できることが知られているが，その記述方法は大きく分けると，関数オーバーロードを利用したもの (FO) と Detection Idiom(DI) [2] の 2 つがあげられる．これらの方法は SFINAE [3] やテンプレートの部分特殊化といった知識が必要とされる．DI をさらに一般化し，型への要求部分とそれを受け取り真理値の取得をするメタ関数 (detector) に分離するメタ関数コールバックと呼ばれる方法 (MFC) も提案されている [2]．要求部分の記述そのものには SFINAE 等の知識を必要としないため detector さえ与えられればそれらの知識を意識することなくメタ関数の記述をすることが可能である．しかし，要求部分の記述には declval [4] 等の無名の値が何度も出現することでコードは肥大化し，複雑になってしまう問題がある．

Simplified code of meta functions for type constraints in C++
Naoya Taki, Koichi Moriyama, Atsuko Mutoh and Nobuhiro Inuzuka
Nagoya Institute of Technology, Nagoya 466-8555, Japan

4　requires 式のアイデアを用いた記述

4.1　requires 式

requires 式 [1] とは C++で提案されている型に対する要求部分の記述を簡潔にした構文である．変数リストで要求する式に必要な変数を宣言することで declval のような冗長な記述を回避できる．また，変数名をつけることで意味をもたせることもできる．ソースコード 1 は requires 式で型に+の演算子を要求する例である．

ソースコード 1: requires 式の例

```
1  template <class T>
2  concept C =  requires (T a, T b) {
3    a + b; //T 型は 2 項演算 + が有効か？
4  };
```

4.2　提案手法

本研究では，requires 式のアイデアを用い，関数テンプレートの引数リストを利用することで変数名を扱える記述方法を提案する．MFC と同様に要求部分と detector に分けることで SFINAE 等の知識を必要としないものとする．要求部分は，ある決められた名前 (本研究では requires とする) のメンバー関数テンプレートを宣言したクラスとする．実際にその関数が評価されることはないので関数定義の必要はない．引数として宣言された変数を，戻り値の型を後置する関数宣言構文と式の型を取得する decltype [4] 内で使用することで型に対する要求を記述することができる．ソースコード 2 にはソースコード 1 と同じ要件での記述例を示す．

Copyright ©2018 Information Processing Society of Japan.
All Rights Reserved.

ソースコード 2: 要求部分の例

```
1 struct Requirement {
2   template<class T>
3   auto requires(T a, T b)->decltype(
4     a + b //T 型は 2 項演算 + が有効か？
5     );
6 };
```

detector は型引数として要求部分と検知したい型を受け取り，要求部分の requires 関数に対して検知したい型での実体化が有効かどうかを調べればよい．その実装には DI 等のような従来の記述方法を用いる．ソースコード 3 では実装例を示す．

ソースコード 3: detector の例

```
1 template<class AlwaysVoid,class R,
      class... Args>
2 struct detector :std::false_type{};
3 template<class R, class... Args>
4 struct detector<
5   std::void_t<decltype(&R::template
      requires<Args...>)>,
6   R,Args...> :std::true_type{};
```

また，マクロを利用することで要求部分の宣言とそれを detector に渡したタイプエイリアスの記述を省略できる．

5 実験と考察

5.1 実験環境

FO，DI，MFC，提案手法 (RE)，提案手法+マクロ (RE+M) の 5 つにおいてコード量の比較とアンケートを行った．コード量の比較は，C++の規格として決められている要件 [4] の中から Iterator と UnorderedAssociativeContainer(UAC) を例に文字数で比較をした．特徴として前者は要求部分が少ないのに対し後者は多い．なお，detector の実装部分は含まないものとする．

アンケートは名古屋工業大学の情報工学科の学生 4 人とコンピュータ部の学生 4 人の計 8 人に対して行った．最初にメタ関数の記述例を紹介した後，ある要件に対するそれぞれの手法でのコードを見てもらい「読みやすさ」と「書きやすさ」の 2 点において各手法の順位付けをしてもらった．

5.2 実験結果・考察

図 1 は各手法でのコード量の比較を，図 2 はアンケート調査の結果を示す．

Iterator はコード量の差は少なく，マクロを使用しない場合，MFC よりもやや多くなった．対して UAC では既存手法の約半分に抑えることができた．アンケートに関して提案手法の 2 つは高い順位ではあるものの，MFC とは大きく差が出なかった．

図 1，2 より RE+M はコード量は一番短くなるもののマクロによって元の構文を隠蔽してしまっているためか RE よりも好まれることはなかった．また，図 2 より FO，DI は SFINAE 等による分岐の記述もあることで複雑に思われたと考えることができる．

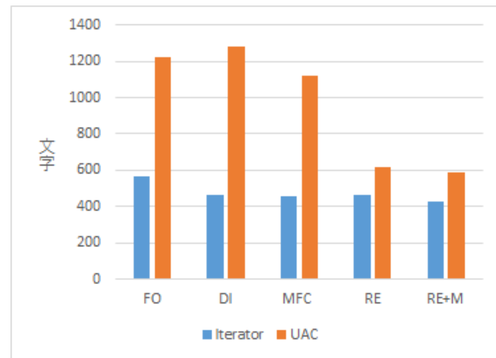

図 1: コード量の比較

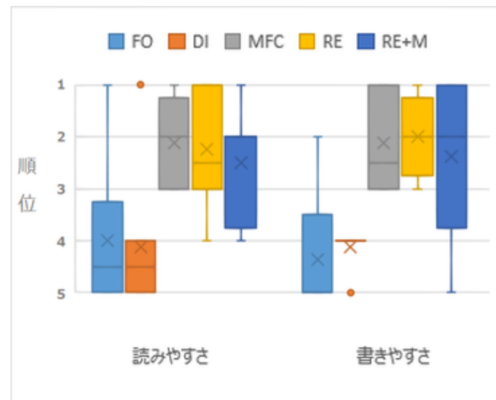

図 2: アンケート調査

6 おわりに

提案手法では requires 式のように変数名を使って型に対する要求の記述をまとめることで，冗長な記述を減らしコード量の削減を可能にすることが確認された．可読性については多くの知識を必要とする記述方法よりは好まれたものの，変数名が扱えない MFC との大きな差は確認することができなかった．今後は被験者の数がより増えた際にアンケート結果にどれだけ変化があるかも確認の必要がある．

参考文献

[1] Andrew Sutton. ”Wording Paper, C++ extensions for Concepts” ISO/IEC JTC1/SC22/WG21 document P0734R0，2017

[2] Walter E. Brown.”Proposing Standard Library Support for the C++ Detection Idiom” ISO/IEC JTC1/SC22/WG21 document N4436，2015

[3] D. Vandevoorde and N. M. Josuttis. C++ Templates: The Complete Guide. Addison-Wesley, 2002

[4] ISO/IEC 14882:2017

Copyright ©2018 Information Processing Society of Japan.
All Rights Reserved.