Designing Expression Templates with Concepts

Bruno Bachelet, Loïc Yon

Designing Expression Templates with Concepts

Bruno Bachelet [1,4], Loïc Yon [2,4]

Research Report LIMOS/RR-15-02

December 3, 2015

1. bruno.bachelet@isima.fr - http://frog.isima.fr/bruno
2. loic.yon@isima.fr - http://www.isima.fr/~loic
3. LIMOS, UMR 6158-CNRS, Université Blaise Pascal, BP 10125, 63173 Aubière, France.

Abstract

Concepts are likely to be introduced in a future C++ standard. They can be used for constraining template parameters, which enables checking requirements on template parameters sooner in the compilation process, and thus providing more intelligible error messages to the user.

They can also be used in the specialization of templates, thus leading to a better control over the selection of the most appropriate version of a template for a given instantiation. This latter aspect offers new possibilities in the design of template libraries, as it enhances the specialization mechanism of templates, and set it up as a solid alternative to inheritance when static binding can replace dynamic binding.

This report addresses the design of expression templates (i.e. templates that represent expressions and are usually built through operator overloading) that are useful to develop an embedded domain specific language (EDSL), and can speed up the evaluation of an expression by delaying the evaluation of intermediate operations to avoid unnecessary temporary objects.

We propose to use concept-based template specialization to parse expression templates in order to ease the design of an EDSL. This approach is a static variant of the well-known visitor design pattern that replaces the overridden methods in the double dispatch of the original design pattern by template specializations based on concepts. An example of EDSL for linear programming developed with our solution demonstrates that a concept-based design helps producing concise and reliable code.

Keywords: generic programming, template specialization, concept-based specialization, template metaprogramming, expression templates.

Résumé

Les concepts seront probablement introduits dans une future norme du C++. Ils servent notamment à contraindre les paramètres d'un patron, ce qui permet de vérifier des exigences sur des paramètres du patron plus tôt dans le processus de compilation, et de fournir ainsi des messages d'erreur plus compréhensibles pour l'utilisateur.

Ils peuvent servir aussi dans la spécialisation des patrons, conduisant alors à un meilleur contrôle de la sélection de la version la plus appropriée d'un patron pour une instanciation donnée. Ce dernier aspect offre de nouvelles possibilités pour la conception de bibliothèques génériques, car il améliore le mécanisme de spécialisation des patrons et le positionne comme une solide alternative à l'héritage quand la liaison statique peut remplacer la liaison dynamique.

Ce rapport aborde la conception de patrons d'expressions (*template expressions*, i.e. des patrons qui représentent des expressions et sont généralement construits par le biais de la surcharge d'opérateurs) qui sont utiles pour développer un langage dédié embarqué (*Embedded Domain Specific Language - EDSL*), et peuvent accélérer l'évaluation d'une expression en retardant l'évaluation d'opérations intermédiaires afin d'éviter des objets temporaires inutiles.

Nous proposons d'utiliser la spécialisation de patron orientée concept pour parcourir les patrons d'expressions afin de faciliter la conception d'un EDSL. Cette approche est une variante statique du célèbre patron de conception *visiteur* qui remplace les redéfinitions de méthodes du double *dispatch* du patron de conception original par des spécialisations de patron reposant sur les concepts. Un exemple d'EDSL pour la programmation linéaire développé avec notre solution démontre que la conception orientée concept aide à produire un code concis et fiable.

Mots clés : programmation générique, spécialisation de patron, spécialisation orientée concept, métaprogrammation par patrons, patrons d'expression.

1 Introduction

Concepts present many advantages for template programming. As they can express constraints on template parameters, they can improve checking the correctness of template use [11]. Without those constraints, when a template is used, no immediate checking is performed on the types bound to the template parameters. And afterward, when the types are effectively manipulated inside the template, errors can be detected, usually with unintelligible messages to the user as they point to the internals of the template.

Concepts can also be used to guide the specialization process of templates, like `enable_if` [4] can achieve in some contexts, but offering new possibilities and safer instantiation [2]. Without concepts, the template specialization mechanism is based on the type pattern of the parameters, which is not always the best way to guide the specialization process: type patterns are missing some information on types that could be relevant to define specializations.

However, concepts reveal to be a complex notion to integrate in C++. After a first attempt for the C++ language to support concepts [7], a second proposal, "Concepts Lite" [13], has been prototyped in GCC [5] and published as an ISO/IEC Technical Specification [12]. This extension introduces template constraints, which is a subset of concepts that allows the use of predicates to constrain template parameters. The long-term goal of this extension is to propose a complete definition of concepts.

In this report, we focus on the use of concepts to enhance template specialization. Several library-based proposals have been made to emulate concepts [10, 11], but with no, or a limited, mechanism for concept-based specialization. In a previous work, we proposed a library-based solution that enables representing partially concepts and using them for template specialization [2]. We choose here to use this library, called C4TS++ [6] (*"Concepts for Template Specialization in C++"*), as it is fully portable (C++03 compliant, with syntactic improvements in C++11), and light (only a small subset of concepts is implemented).

We consider here the design of embedded domain specific languages (EDSL) using concept-based specialization to show some new possibilities of concepts. In C++, EDSL's use operator overloading to propose a language suited for a specific domain (e.g., linear algebra with matrix and vector operations). Besides the syntactic aspect, the underlying metaprogramming technique called "expression templates" enables speeding up the evaluation of an expression (notably, it allows evaluating an expression in a single pass that avoids temporary objects) [15].

Evaluation means parsing the expression and visit each one of its operations and operands to perform a specific action. A generic approach for evaluating an expression is proposed here, based on a static variant of the well-known visitor design pattern [5], that uses concept-based specialization. Like the original design, this solution implements a double dispatch mechanism that makes the solution open for extension: new kinds of visits and operands can be added at will. Concepts also help making the code more reliable and concise, as visits are ruled by concepts instead of type patterns.

Section 2 introduces notions on concept-based specialization and the syntax used all along the document. Section 3 briefly recalls the expression templates technique and its benefits. Section 4 presents our concept-based design [7] to build and evaluate expression templates with a static variant of the visitor design pattern. Section 5 shows how to build an EDSL for linear programming with our solution.

4. `http://en.cppreference.com/w/cpp/types/enable_if`
5. `http://concepts.axiomatics.org/~ans/`
6. `http://forge.clermont-universite.fr/projects/cpp-concepts`
7. Source code is available at: `http://forge.clermont-universite.fr/projects/et-concepts`

2 Concept-Based Specialization

2.1 Concepts and Relationships

When a type is bound to a template parameter, it must fulfill some requirements from the template. These requirements can be represented by a concept that defines syntactic constraints (i.e., on the interface of the type) and semantic constraints (i.e., on the behavior of the type). When a type fulfills the requirements of a concept, it is said that the type "models" the concept. The notion of a specialization between concepts is called "refinement": a concept that includes the requirements of another concept is said to "refine" this concept.

For instance, let us define the concept `Integral` that captures the requirements of an integral number, and the concept `Numerical` that captures the requirements of any kind of number. One can state that type `int` models concept `Integral`, and concept `Integral` refines concept `Numerical`.

This taxonomy of concepts can be used for template specialization: one can provide a template `Example<T>` with specializations for `Numerical` and for `Integral`. Depending on the type bound to `T`, the most specialized version of the template must be selected: version `Integral` for `T` = `int`, version `Numerical` for `T` = `double`. Notice that `int` also models `Numerical`.

2.2 C4TS++ Library

Waiting for concepts to be part of C++ standard, we proposed a solution to allow template specialization based on concepts [2]. Due to portability concerns, our goal was to provide a purely library-based solution that could be used with any standard C++ compiler, and no need of an additional tool. This library is based on template metaprogramming techniques, and uses macros only as front-end to provide a light syntax to the user. It is C++03 compliant, but its interface has recently been completed using C++11 features to simplify the syntax [8]. Nevertheless, the core of the library remains as presented in [2].

This library, called C4TS++, provides syntax to declare concepts, modeling relationships and refinement relationships. Based on these declarations, template specialization with concepts can be achieved. Concepts are used to constrain parameters in a specialization. At instantiation time, the most appropriate version of a template is selected based on the concepts modeled by the types bound to the parameters: a metaprogram determines, for each one of these types, the most specialized concept to consider for this instantiation, based on the declared taxonomy of concepts. This solution is also open for extension: new concepts, relationships, and template specializations can be defined at any time; such additions will then be picked up by the specialization mechanism.

2.3 Code Example

The example of Section 2.1 is written here using C4TS++. First of all, concepts have to be declared, using macro `gnx_declare_concept`.

```
gnx_declare_concept(Numerical);
gnx_declare_concept(Integral);
gnx_declare_concept(Floating);
```

8. Since version 2015-02-27.

Modeling and refinement relationships can be added, using macro `gnx_add_models`. Notice that both kinds of relationships are declared using the same instruction.

```
template <> gnx_add_models(Integral,Numerical); // Refinement
template <> gnx_add_models(Floating,Numerical); // Refinement

template <> gnx_add_models(int,Integral);    // Modeling
template <> gnx_add_models(double,Floating); // Modeling
```

From now on, a taxonomy of concepts is defined, and it can be extended at any time through new concept and/or relationship declarations (providing that a few rules are followed to prevent some "Schrödinger's Cat" effect with templates [9], cf. technical report [3]).

Defining template specializations based on concepts is simple, but the primary version of the template must be prepared. First, we need an identifier to represent a case of template definition with specializations, which is called a "specialization context" (the reason is that our mechanism needs to know all the concepts involved in the specializations of a given template; as code analysis is not possible with a library-based approach, the solution is to make explicit declarations that associate concepts to a specialization context). The context must be unique for each case of template definition; it can be an existing type, or a type specially declared for this purpose.

```
struct ExampleContext;
```

Second, for each template parameter `T` that will be constrained by concepts in specializations, an additional parameter `C` is necessary to represent the most appropriate concept of `T` in this context of specialization (the value of `C` will be deduced automatically by metafunction `gnx_best_concept_t`).

```
template < class T,
           class C = gnx_best_concept_t<ExampleContext,T>
         >
struct Example;
```

The template is now prepared for concept-based specialization: the additional parameter can be constrained by any concept to define a specialization, the only requirement being to declare the use of the concept to the context (using macro `gnx_add_uses`).

```
template <> gnx_add_uses(ExampleContext,Numerical);

template <class T> struct Example<T,Numerical>
{ Example(void) { cout << typeid(T).name() << " = Numerical" << endl; } };

template <> gnx_add_uses(ExampleContext,Integral);

template <class T> struct Example<T,Integral>
{ Example(void) { cout << typeid(T).name() << " = Integral" << endl; } };

template <> gnx_add_uses(ExampleContext,Floating);

template <class T> struct Example<T,Floating>
{ Example(void) { cout << typeid(T).name() << " = Floating" << endl; } };
```

New concepts and/or relationships can be added at any time. They will be considered in the specialization process as long as the template has not been instantiated (cf. technical report [3]). For instance, a class `MyNumber` can be defined and declared to model `Numerical`, and automatically the `Numerical` version of the template will be selected when instantiating `Example<MyNumber>`.

```
class MyNumber { [...] };

template <> gnx_add_models(MyNumber,Numerical);
```

9. Metaphor from `http://www.codeproject.com/Articles/776770/Automatic-Static-Counter`.

3 Expression Templates

Expression templates are a technique introduced by [15] and [14] to represent an expression as an object, using templates to build the type of this object. The main goal of expression templates is to tackle performance problems that may occur with operator overloading. The structure of an expression is represented by a recursive composition of types that models an abstract syntax tree (AST): an expression is an operation on operands that are expressions. With C++11, the composition can be modeled by a single template `Expression` with a parameter to represent the operator characterizing the operation, and a pack of parameters to represent the operands.

```
template <class OPERATOR,class... OPERANDS> struct Expression {
 std::tuple<const OPERANDS &...> operands;

 template <class... OPS> explicit Expression(OPS &&... ops)
 : operands(ops...) {}

 double evaluate(unsigned i) const
 { return OPERATOR::evaluate(operands,i); }
};
```

The code presented in this section is simplified to point out the basics only (notably, type passing should be optimized with perfect forwarding, and types should be stripped of qualifiers to bind the template parameters of `Expression`). Assume now that each unary or binary arithmetic operator of C++ is represented by a class defined according to the following pattern.

```
struct AdditionOperator {
 template <class OP1,class OP2>
 static double evaluate(const std::tuple<OP1,OP2> & tuple,unsigned i) {
  return std::get<0>(tuple).evaluate(i) + std::get<1>(tuple).evaluate(i);
 }
};
```

Consider a template `Array<N>` (similar to `std::array`) that represents an array of size `N`, we would like to overload the arithmetic operators so operations on arrays are applied on each element, e.g., operation `c = a+b` means `c[i] = a[i]+b[i]` for each element at index `i`. Expression `-a+b*c` for instance, where variables `a`, `b` and `c` are objects of class `Array<N>`, can be modeled by a recursive composition using template `Expression`.

```
using exp_t = Expression< AdditionOperator,
                          Expression< MinusOperator,
                                      Array<N>
                                    >,
                          Expression< MultiplicationOperator,
                                      Array<N>,
                                      Array<N>
                                    >
                        >;
```

Notice that, at compile time, such a static structure could be parsed using metaprogramming techniques to generate a specific code. Assume now that arithmetic operators have been overloaded as follows.

```
template <class OP1,class OP2>
inline Expression<AdditionOperator,OP1,OP2> operator+(OP1 && op1,
                                                      OP2 && op2)
{ return Expression<AdditionOperator,OP1,OP2>(op1,op2); }
```

This overloading allows code `-a+b*c` to automatically produce an object of type `exp_t`. Before introducing lambda expressions in C++11, such objects could be used to represent lambda

functions [15], but the main interest of expression templates is their ability to delay the evaluation of intermediate operations to avoid unnecessary temporary objects. For instance, let us consider the assignment `d = -a+b*c`, where `d` is an object of class `Array<N>` and template `Array` has the following assignment operator.

```
template <class EXPRESSION>
inline Array<N> & Array<N>::operator=(const EXPRESSION & expression) {
 for (unsigned i = 0; i<N; ++i) values[i] = expression.evaluate(i);
 return *this;
}
```

The AST of the expression is built and passed as argument to the assignment operator. In this function, method `evaluate` is recursively called on every operation and operand of the expression (notice that `Array` must have a method `evaluate`), which results in a single loop and the inlining of the whole evaluation. The code generated for the assignment operator has a performance equivalent to:

```
for (unsigned i = 0; i<N; ++i) d[i] = -a[i] + b[i]*c[i];
```

Notice that basic operator overloading would create temporary objects during the evaluation of expression `-a+b*c`, which would produce a code having a performance equivalent to:

```
Array<N> a1; for (unsigned i = 0; i<N; ++i) a1[i] = -a[i];
Array<N> a2; for (unsigned i = 0; i<N; ++i) a2[i] = b[i]*c[i];
Array<N> a3; for (unsigned i = 0; i<N; ++i) a3[i] = a1[i]+a2[i];
```

4 Expression Templates with Concepts

We propose to use concepts to design a framework for expression templates, with the aim of modeling expressions and overloading operators once and for all, so users can focus on expression parsing. In the previous example, template `Expression` is designed for a single kind of evaluation (cf. `evaluate` method), whereas one might need various kinds of evaluation (e.g., computation, display, semantic analysis...). Our solution uses concept-based specialization as a reliable and extensible way of defining evaluations. It is inspired from the double dispatch of the visitor design pattern [5], with template specialization replacing method overriding. Concepts could also provide more control over operands, e.g., static assertions formulated based on concepts could detect the use of wrong operands for an operation.

First, our design for expression templates is introduced, with a specific care on the possible storage of an expression to delay evaluation. Then, the taxonomy of concepts that will guide the parsing of expressions, and operator overloading that will produce expression objects, are presented. Finally, our solution to evaluate an expression with concept-based specialization is detailed.

4.1 Modeling

A different version of template `Expression` is proposed, with no evaluation method (the process is externalized, cf. double dispatch), and a new template parameter (boolean `FIXED`) to anticipate the possibility of storing an expression to postpone evaluation. Parameter `FIXED` indicates whether an expression is an "rvalue" ("transient" expression, temporary object destroyed at the end of instruction), or an "lvalue" ("fixed" expression, a copy of the temporary version made for further use) [10].

10. `http://en.cppreference.com/w/cpp/language/value_category`

```
template <bool FIXED,class OPERATOR,class... OPERANDS>
struct Expression : public AbstractExpression {
 using operator_t = OPERATOR;
 using operands_t = std::tuple<OPERANDS...>;

 std::tuple<etc_transient_operand_t<OPERANDS>...> operands;

 template <class... OPS> explicit Expression(OPS &&... ops)
 : operands(std::forward<OPS>(ops)...) {}
};
```

The primary version of template `Expression` represents a transient expression (i.e., `FIXED = false`). As shown in Figure 1, the template models concept `cExpression` that imposes features to access the operator and the operands of the expression. Notice the use of metafunction `etc_transient_operand_t`[11] that opens the possibility of choosing the way of storing an operand (by reference, the default choice to avoid unnecessary copies, or by copy).

4.2 Fixing an Expression

One might need to store an expression for further use: to postpone the evaluation, or to perform multiple evaluations in sequence (e.g., analyzing the AST for parallel evaluation). Let us consider such an example, where arithmetic operators are overloaded to return transient expressions (as detailed in Section 4.4).

```
auto & e = -a+b*c;
[...]
f(e);
```

This code is incorrect, as the AST built for the expression is an rvalue, which makes reference `e` invalid at the last line. The expression must be copied to get an lvalue. Template `Expression` is specialized for `FIXED = true` to represent a fixed expression. Its constructor makes possible the copy of a transient expression to get a fixed expression: the conversion of each operand is made by metafunction `etc_fixed_operand_t<T>` that returns the type to use for fixing and storing an operand of type `T`.

```
template <class OPERATOR,class... OPERANDS>
struct Expression<true,OPERATOR,OPERANDS...> : public AbstractExpression {
 using operator_t = OPERATOR;
 using operands_t = std::tuple<OPERANDS...>;

 std::tuple<etc_fixed_operand_t<OPERANDS>...> operands;

 template <class... OPS>
 Expression(const Expression<false,OPERATOR,OPS...> & expression)
 : operands(expression.operands) {}
};
```

Operands that are rvalues must be copied (mainly the instances of template `Expression`), but not necessarily operands like `a`, `b` and `c` that are lvalues. Depending on the application, one might want to keep references or make copies of operands `a`, `b` and `c`. The default behavior is that expressions keep constant references of lvalues, but metafunction `etc_fixed_operand_t` can be specialized.

11. We chose to prefix all the metafunctions and macros of our library with "`etc_`".

Function `etc_fix` is provided to help (thanks to type deduction) the copy of a transient expression of type `T` to make a fixed expression of type `etc_fixed_t<T>`. The previous example must be corrected as follows.

```
auto e = etc_fix(-a+b*c);
[...]
f(e);
```

A template `Literal<FIXED,TYPE>` is necessary to keep track of rvalue operands of type `TYPE` that are not objects based on template `Expression`. For instance, in expression `3*a`, the first operand is an rvalue that must be copied when the whole expression is being fixed. Operator overloading is designed to automatically encapsulate such an rvalue with template `Literal` (cf. "boxing" in Section 4.4). For expression `3*a`, it produces an object of the following type.

```
Expression< false,MultiplicationOperator,
            Literal<false,int>,
            Array<N>
          >
```

Template `Literal` is considered a unary expression: it models concept `cLiteral` that refines concept `cExpression` (cf. Figure 1). Therefore, it provides the same features as template `Expression`. It is merely a wrapper for an operand that is either transient (it keeps a constant reference of the operand) or fixed (it keeps a copy of the operand).

4.3 Taxonomy of Concepts

As parsing an expression is based on concepts, it is necessary to define a taxonomy of the concepts that characterize the operations and operands of expressions.

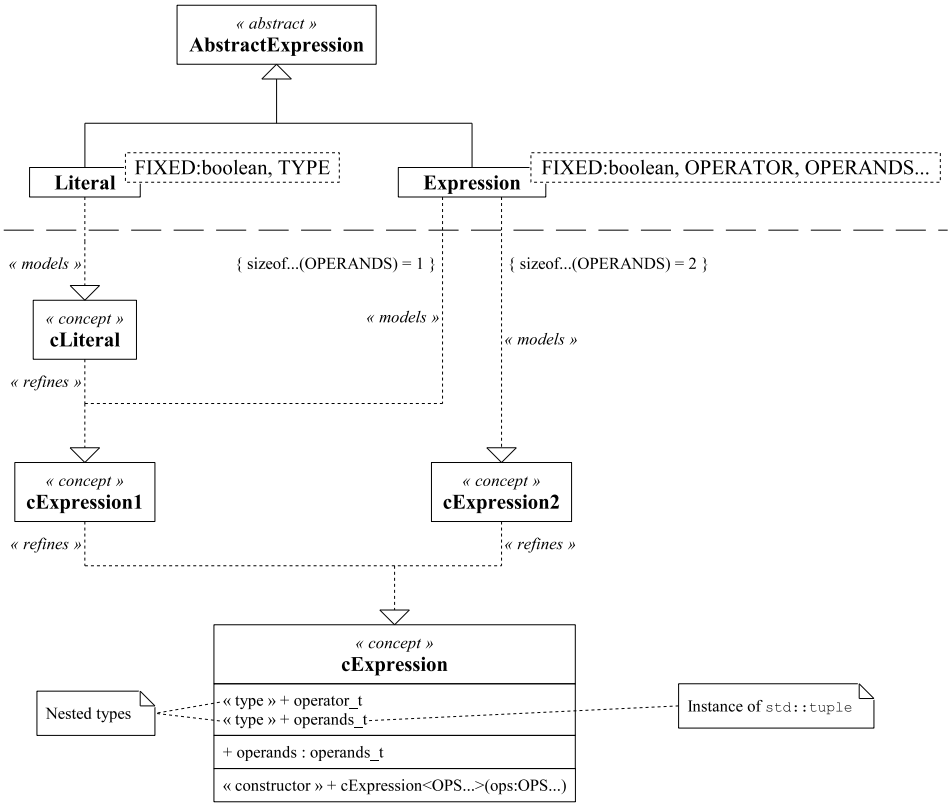

Figure 1: Taxonomy of concepts for expressions.

Figure 1 shows that templates `Expression` and `Literal` indirectly model concept `cExpression`, and that expressions are classified according to their arity: concept `cExpression1` for unary expressions, concept `cExpression2` for binary expressions... Both templates also inherit from abstract class `AbstractExpression` to enable type erasure (cf. Section 4.5.3).

Each operator of interest for expression templates is represented by a class (like `AdditionOperator` presented in Section 3, but with no evaluation method), and the associated operation is characterized by a concept. The taxonomy of concepts is completed so expressions based on the operator model the concept of the operation.

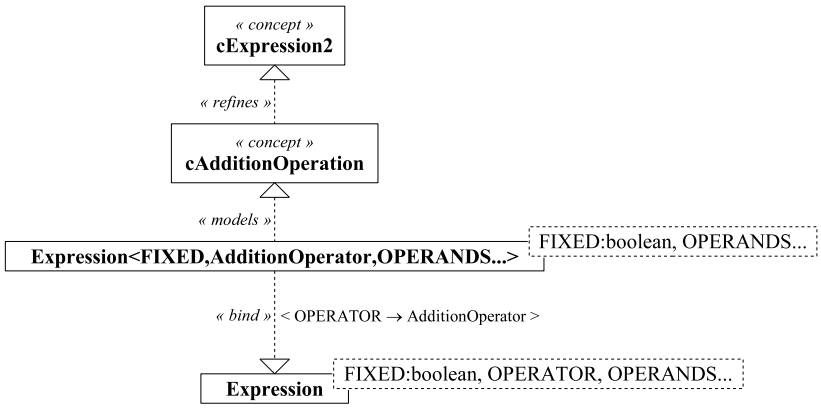

Figure 2: Addition operation in the taxonomy of concepts.

For instance, declaring the addition operator implies defining class `AdditionOperator` and declaring concept `cAdditionOperation` that refines `cExpression2` (cf. Figure 2). Besides, any instance of `Expression` with parameter `OPERATOR` = `AdditionOperator` models concept `cAdditionOperation`. This declaration can be achieved as follows using C4TS++.

```
struct AdditionOperator;

gnx_declare_concept(cAdditionOperation);

template <> gnx_add_models(cAdditionOperation,cExpression2);

template <bool FIXED,class... OPERANDS>
gnx_add_models(Expression<FIXED,AdditionOperator,OPERANDS...>,
               cAdditionOperation);
```

The taxonomy can be extended, notably to detect the original semantics of an operator, e.g., to know if it is arithmetic, logical or relational. For this purpose, concepts `cArithmeticOperation`, `cLogicalOperation` and `cRelationalOperation` have been inserted in the taxonomy (e.g., concept `cAdditionOperation` refines concept `cArithmeticOperation`).

4.4 Operators Overloading

Once an operator has been declared, the associated function must be overloaded to return an expression object. For instance, function `operator+` has to be overloaded to return an object of type `Expression<false,AdditionOperator,B1,B2>`, where `B1` and `B2` are the types of the two operands. However, such an overload requires some care, as shown in the following code (based on this pattern, macros are provided to allow overload with a single instruction [12]).

12. Macros `etc_overload_operator[1|2]` are provided to declare a unary or binary operator, and overload the associated function in a single line: `etc_overload_operator2(Addition,operator+,"+")`.

```
template < class OP1,class OP2,
           class T1 = gnx_base_type_t<OP1>,
           class T2 = gnx_base_type_t<OP2>,
           class B1 = etc_operand_boxing_t<OP1 &&>,
           class B2 = etc_operand_boxing_t<OP2 &&>
         >
typename enable_if< gnx_or< etc_is_operand<T1>,
                            etc_is_operand<T2>
                          >,
                    Expression<false,AdditionOperator,B1,B2>
                  >::type
operator+(OP1 && op1,OP2 && op2) {
 return Expression<false,AdditionOperator,B1,B2>(std::forward<OP1>(op1),
                                                 std::forward<OP2>(op2));
}
```

The overload is parameterized on types `OP1` and `OP2` (deduced at function call) that are converted into `B1` and `B2` (operation called "boxing") using metafunction `etc_operand_boxing_t`. This metafunction is designed to convert any rvalue that is not based on template `Expression` into an instance of template `Literal` (as explained in Section 4.2). `OP1` and `OP2` are also stripped of their qualifiers to get `T1` and `T2` (e.g., `const int &` becomes `int`) using metafunction `gnx_base_type_t` (note that it is also used in boxing to get types `B1` and `B2` without qualifiers).

Without constraints on `OP1` and `OP2`, the overload would be valid for any type of operand. The SFINAE principle of C++, through template `enable_if` [4], is applied to get control: the overload is valid only if `T1` or `T2` is a type activated to be an operand (metafunction `etc_is_operand<T>` returns whether type `T` is activated). Initially, only types based on templates `Expression` and `Literal` are activated to be operands, and one has to specialize metafunction `etc_is_operand` to formally activate a type. Macro `etc_activate_operand` is provided to help the specialization. For instance, `Array<N>` can be activated as follows.

```
template <unsigned N> etc_activate_operand(Array<N>);
```

4.5 Expression Evaluation

4.5.1 Visitor Design Pattern

The visitor design pattern is a well-known solution to represent an operation to apply on each element of an heterogeneous set, where the code of the operation depends on the type of the element [5]. The pattern allows defining new operations with no impact on the classes of the elements. In our case, it could be used to evaluate an expression (cf. Figure 3), but it presents many restrictions: it is based on dynamic binding (which could lead to significant execution time overhead), and elements must belong to the same base class (which narrows the possibilities of extension, as any object cannot be an element in this pattern).

The principle is that an object, the visitor, is moved from one element to the next to perform the operation. The code executed for each visit depends on the type of the visitor (i.e., the kind of operation to perform) and on the type of the element. This dispatch is based on two virtual methods: `accept` that is overridden for each type of element (to redirect to the correct `visit` method of the visitor), and `visit` that is overloaded for each type of element and is overridden for each kind of operation (cf. Figure 3, where the visited elements are operands of an expression). A visit is achieved by calling the `accept` method of the element with the visitor as argument.

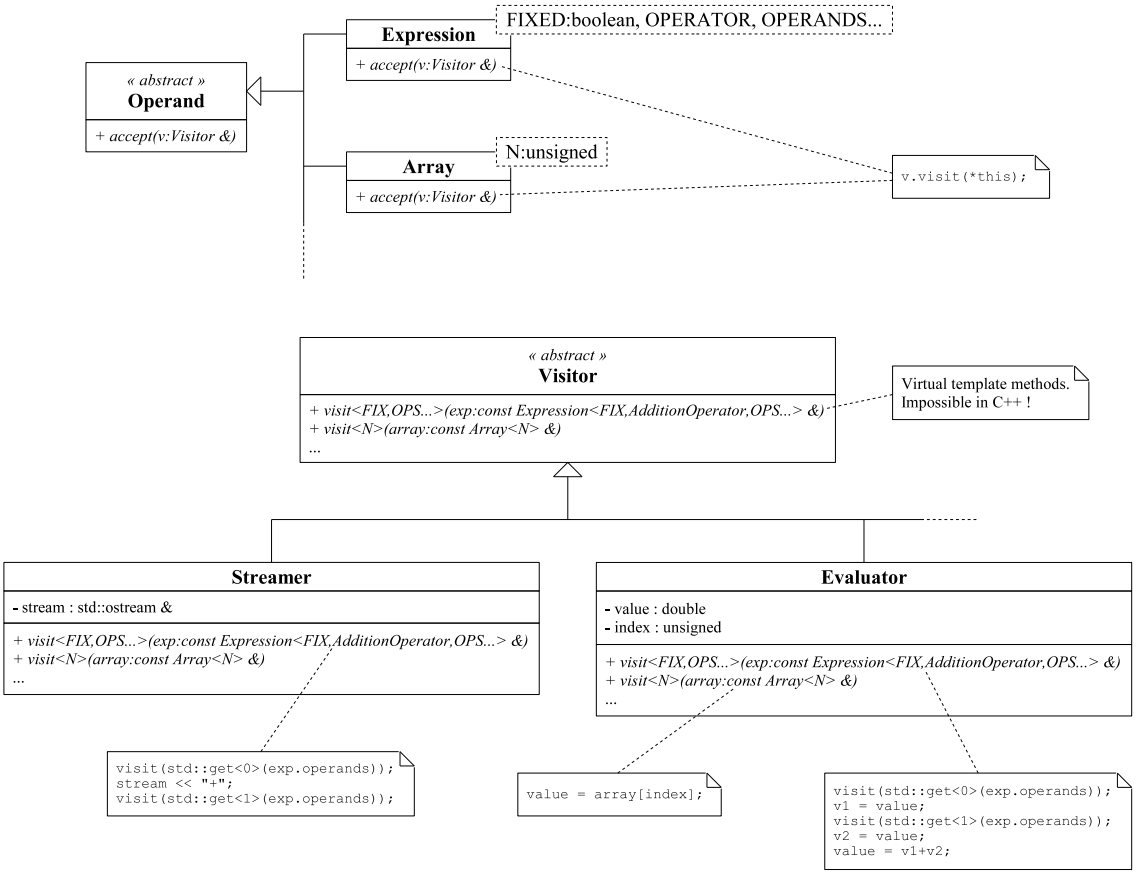

Figure 3: Visitor design pattern for expression evaluation.

In this design, the elements belong to the same base class, whereas with expression templates, the elements can be of various types with no class relationship, such as the instances of templates `Expression` and `Array`. Figure 3 also presents methods `visit` as virtual template methods, which is impossible in C++. Therefore, the `visit` method should be explicitly overloaded for possibly any instance of template `Expression`, which is hardly tractable. Besides, the signature of method `visit` is fixed: in our example, there is no value returned and a single argument (the operand). A workaround is to add attributes to the visitor to represent the return value and the arguments, but it could lead to unclear code (cf. visitor `Evaluator`).

4.5.2 Concept-Based Visitor

A solution with concepts is proposed that keeps the idea of double dispatch of the visitor design pattern. This static approach is implemented by a template `etc_visit` with two parameters: the type of the visitor and the type of the operand.

```
template < class VISITOR,
           class OPERAND,
           class CONCEPT = gnx_best_concept_t<VISITOR,OPERAND>,
           class ENABLE = void
         >
struct etc_visit;
```

The double inheritance of the original design pattern is replaced by a single template specialization: for instance, instead of specializing class `Visitor` with class `X` and class `Operand`

with class `Y`, template `etc_visit` is specialized with `VISITOR = X` and `OPERAND = Y`. The specialization process must be controlled by concepts, as presented in Section 2, which requires an additional parameter `CONCEPT`. A last parameter `ENABLE` is also added to help applying SFINAE if necessary.

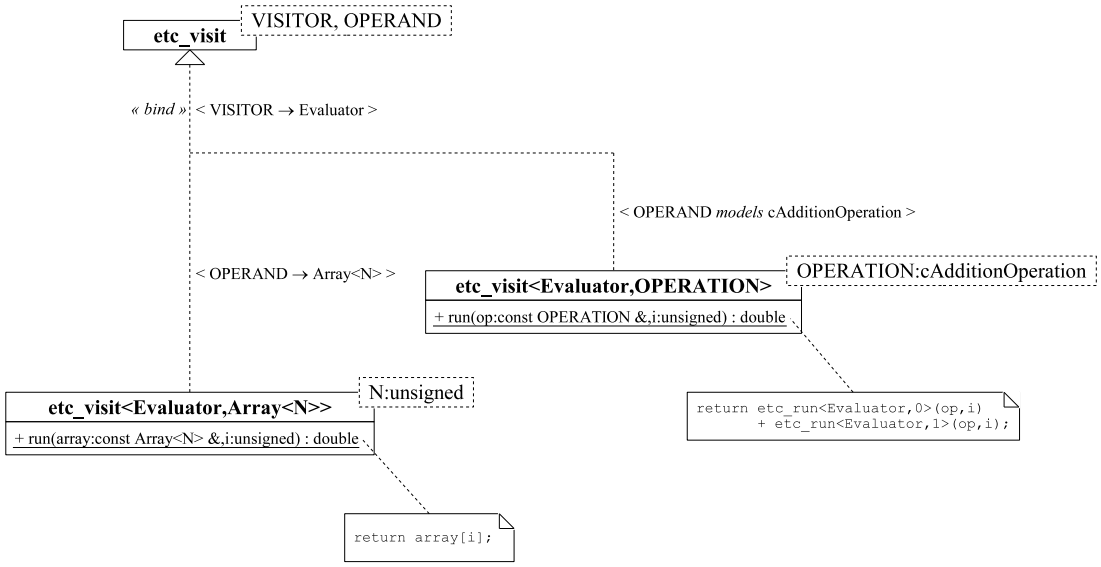

Figure 4: Concept-based visitor for expression evaluation.

The notion of visitor remains in the form of a class acting as identifier of the type of operation applied on the elements, but the instantiation of an object is not mandatory. Let us declare an empty class `Evaluator` to represent the operation of computing the element at index `i` of an expression of `Array<N>` objects. For instance, to define the visit of the addition operator, template `etc_visit` is specialized for concept `cAdditionOperation` and visitor `Evaluator` (cf. Figure 4 and the following code). Notice that the visitor class also acts as specialization context (cf. Section 2).

```
template <> gnx_add_uses(Evaluator,cAdditionOperation);

template <class OPERATION>
struct etc_visit<Evaluator,OPERATION,cAdditionOperation> {
 static double run(const OPERATION & op,unsigned i) {
  return etc_run<Evaluator,0>(op,i) + etc_run<Evaluator,1>(op,i);
 }
};
```

The code of the visit is located in a static method `run` of the specialized version of template `etc_visit`. The prototype of this method is almost free, the only constraint being that the first argument must be the operand to process. Parsing all the operands and operations of an expression is recursive: for instance, visiting an addition operation means adding the results of the visits on each operand. A visit is called through function `etc_run` that helps instantiating template `etc_visit`, as shown in the following code.

```
template <class VISITOR,unsigned N,class EXPRESSION,class... ARGS>
inline auto etc_run(EXPRESSION && expression,ARGS &&... args) {
 return etc_visit< gnx_base_type_t<VISITOR>,
                   etc_operand_type_t<EXPRESSION,N>
                 >::run(etc_operand_value<N>(expression),
                        gnx_forward<ARGS>(args)...);
}
```

Functions `etc_operand_type_t` and `etc_operand_value` are provided to ease the access to the type and value of an operand from its position in an expression. Several versions of function `etc_run` are implemented to allow more flexibility. Notably, the function can be called with the index of the operand to visit, as in the previous example, or without index to visit the whole expression. For instance, the assignment operator of template `Array` can be overloaded as follows to fully evaluate the expression received as argument.

```
template <class EXPRESSION>
inline Array<N> & Array<N>::operator=(const EXPRESSION & expression) {
 for (unsigned i = 0; i<N; ++i)
  values[i] = etc_run<Evaluator>(expression,i);

 return *this;
}
```

4.5.3 Expression and Visitor Types Erasure

In all previous examples, expressions are evaluated knowing their exact type at compile time, which limits the performance overhead of expression templates. However, one could delay the evaluation enough to have to store the expression with type erasure (i.e., by losing the concrete type of the expression at compile time). A unique abstract type is then necessary to represent any expression: `Expression` and `Literal` templates both inherit from `AbstractExpression` (cf. Figure 1). This way, any expression can be manipulated as a pointer or a reference of type `AbstractExpression`.

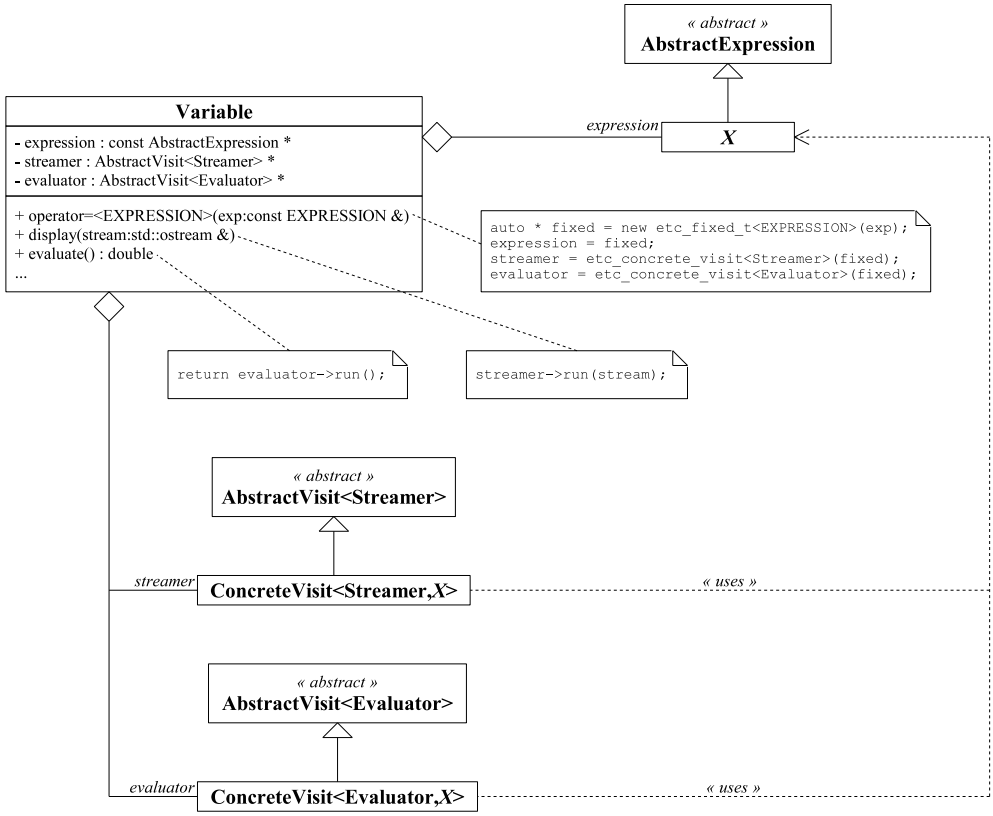

Figure 5: Type erasure to delay expression evaluation.

Let us consider the following example where expression templates are used to represent formulas to compute the values of variables.

```
Variable a,b,c,d,e;

e = a+d;
a = 3;
d = b*c;
b = 7;
c = 14;

cout << e.evaluate();
```

The formulas are written with no specific order: for instance, the formula of variable `d` that depends on `b` and `c` is defined before the formulas of `b` and `c`. The evaluation of `b*c` cannot be performed at the assignment to `d`. It must be delayed after the assignment of `b` and `c`. Thus, each `Variable` object needs to store the expression of its formula with type erasure, because the type of the expression cannot be deduced at the definition of class `Variable`. At assignment, the expression is dynamically copied to get a fixed version that is referenced by a pointer attribute in class `Variable` (cf. Figure 5).

We now consider the possibility to compute the value of a variable (cf. method `evaluate`) and to output its formula (cf. method `display`). For this purpose, visitors `Evaluator` and `Streamer` are defined and used to specialize template `etc_visit`. But to visit an expression, i.e., when calling function `etc_run`, the visitors must be handled with the concrete type *X* of the expression. In our example, type *X* is known at assignment and lost after.

Template class `ConcreteVisit<VISITOR,EXPRESSION>` is defined to represent objects that embed, in their method `run`, the code to visit an object of concrete type `EXPRESSION` with `VISITOR` (cf. Figure 6). These objects are dynamically created during the assignment (by calling the helper function `etc_concrete_visit`), and stored in attributes (cf. `evaluator` and `streamer`), while the concrete type *X* of the expression is known (cf. Figure 5). Afterward, their `run` method can be called to start a visit (cf. methods `evaluate` and `display`).

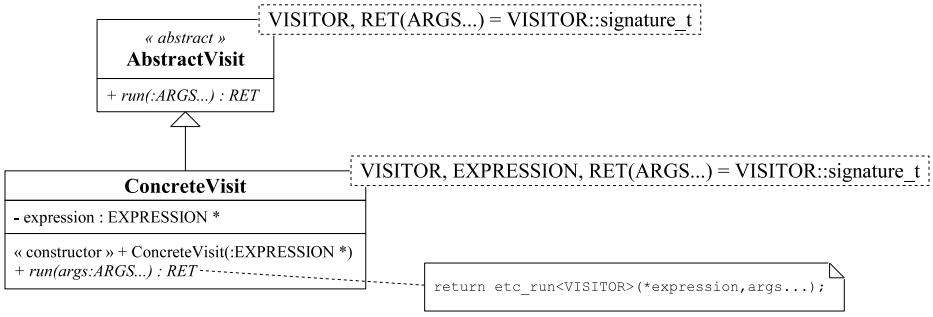

Figure 6: Visit abstraction for type erasure.

As for expressions, type erasure is necessary for `ConcreteVisit` objects. Therefore, abstract class `AbstractVisit<VISITOR>` is defined to represent the visit by `VISITOR` of any expression, so `ConcreteVisit` objects can be manipulated as pointers or references of type `AbstractVisit<VISITOR>`. Notice that method `run` of these objects has to be virtual, which implies a performance overhead when calling for a visit. However, the performance of the visit itself remains unchanged.

To define templates `AbstractVisit` and `ConcreteVisit` requires the signature of the visitor, meaning the return type `RET` and the arguments types `ARGS...` of method `run` in specializations of `etc_visit` for this visitor. To facilitate the instantiation of those templates, a member

type `signature_t` has to be defined in each visitor to represent its signature (using template specialization, types `RET` and `ARGS...` can be deduced from `signature_t` automatically). For instance, visitors `Evaluator` and `Streamer` are defined as follows.

```
struct Evaluator { using signature_t = double(void); };
struct Streamer  { using signature_t = void(std::ostream &); };
```

5 Example of EDSL

An EDSL for linear programming is defined here with concept-based expression templates [13]. A linear program (class `Program`) is a problem where an objective has to be optimized (minimizing or maximizing a linear expression) under constraints that are linear expressions bounded by an inferior or superior value, or equal to a value. A linear expression (class `Linear` [14]) is a weighted sum of variables (class `Variable`). Here is an example of linear program expressed with our EDSL.

$$P \begin{cases} \text{maximize } 3x_1 - 2x_2 + 8x_3 \\ \text{under } 5x_1 - 2x_2 + 4x_3 \leq 8 \\ \qquad x_1 + 3x_2 + 8x_3 \geq 25 \\ \qquad 9x_1 + 6x_2 - 3x_3 = 17 \end{cases}$$

```
Program p;

auto & x1 = p.newVariable();
auto & x2 = p.newVariable();
auto & x3 = p.newVariable();

p.maximize(3*x1 - 2*x2 + 8*x3);

p += 5*x1 - 2*x2 + 4*x3 <= 8;
p += x1 + 3*x2 + 8*x3 >= 25;
p += 9*x1 + 6*x2 - 3*x3 == 17;
```

Class `Variable` is activated to be an operand, which automatically makes any expression with at least one operand of this type to produce an object based on template `Expression`. The latter can be evaluated by visitor `Builder` to build the linear expression (or constraint) that it represents (i.e. a `Linear` object). Notably, the assignment operator of class `Linear` is overloaded the following way.

```
template <class EXPRESSION>
inline Linear & Linear::operator=(const EXPRESSION & expression) {
 clear();
 etc_run<Builder>(expression,*this);
 return *this;
}
```

The visit consists in building progressively a linear expression (or constraint) by associating a coefficient to each variable. The process starts with an expression where all the coefficients are null, and progressively, for each encountered operand, these coefficients are updated, leading finally to a simplified linear expression. For instance, the following constraints will be simplified at their evaluation.

```
3*x1 + 4*(2*x2 - 3*x3) <= 13  ⟶  3*x1 + 8*x2 - 12*x3 <= 13
2*x1 - 3*x2 >= 5*x1 + 2*x3    ⟶  -3*x1 - 3*x2 - 2*x3 >= 0
```

The visit of `Builder` is defined from the specializations of template `etc_visit` described in Table 1. Some are constrained by the type of the operand (e.g., `Variable` and `Linear`), whereas others are constrained by the concepts modeled by the operand (e.g., the addition and multiplication operations).

13. Source code is available at: `http://forge.clermont-universite.fr/projects/et-concepts`
14. To avoid confusion with template `Expression`, class `lin::Expression` from code is called `Linear` here.

Specialization constraints	Assertions
`OPERAND` = `Variable`	
`OPERAND` = `Linear`	
`OPERAND` models `cLiteral`	The literal is arithmetic.
`OPERAND` models `cMultiplicationOperation`	One operand is an arithmetic literal and the other one is a linear expression.
`OPERAND` models `cDivisionOperation`	The left operand is a linear expression and the right operand is an arithmetic literal.
`OPERAND` models `cPlusOperation` or `cMinusOperation`	The operand is a linear expression.
`OPERAND` models `cAdditionOperation` or `cSubtractionOperation`	Both operands are linear expressions.
`OPERAND` models `cRelationalOperation`	Both operands are linear expressions and the relationship is an inferiority, superiority or equality.

Table 1: Specializations for the visit of a linear expression or constraint.

For each visit, assertions are set in order to verify the syntax of the expression and make sure that it is really linear. These assertions are only superficial, because the recursivity of the visit allows in-depth verification. Here is the example of the visit of a subtraction operation.

```
template <> gnx_add_uses(Builder,cSubtractionOperation);

template <class TYPE>
struct etc_visit<Builder,TYPE,cSubtractionOperation> {
 static_assert(is_linear<etc_operand_type_t<TYPE,0>>::value,
               "Left operand must be a linear expression.");

 static_assert(is_linear<etc_operand_type_t<TYPE,1>>::value,
               "Right operand must be a linear expression.");

 static void run(const TYPE & operation,Linear & linear,
                 double coef = 1.0) {
  etc_run<Builder,0>(operation,linear,coef);
  etc_run<Builder,1>(operation,linear,-coef);
 }
};
```

Assertions ensure that both operands are linear expressions: metafunction `is_linear` returns whether the operand is linear by checking that its type is `Linear` or `Variable`, or that it models the concepts of operations supposedly linear such as `cAdditionOperation`, `cMultiplicationOperation`... This test relies on metafunction `gnx_matches` [15] provided by C4TS++.

```
template <class TYPE>
struct is_linear : gnx_matches<Builder,TYPE,
                               Linear,Variable,
                               cAdditionOperation,
                               cMultiplicationOperation,
                               [...]
                              > {};
```

With only 8 specializations [16], verification, building and simplification of a linear expression or constraint are defined. Thanks to concepts, the visit of the expression is fully controlled: the selection of a specialized version of template `etc_visit` is only possible if the operand models the specified concept, and assertions allow additional controls to validate the syntax.

15. The second parameter is compared with the types and concepts that follow, the first one is the observer (cf. Section 2).

16. C4TS++ allows using logical combinations of concepts to control template specialization [1].

6 Conclusion

A new design of expression templates based on concepts is proposed in this report. A taxonomy of the concepts that characterize the operations and operands of expressions has been defined. As it relies on the implementation of concepts by the C4TS++ library, the taxonomy can be extended at will, notably to distinguish operations based on semantics. The evaluation of an expression is a recursive process that goes through the abstract syntactic tree of the expression to apply a specific action on each node. Nodes are operations and operands that can be distinguished based on the concepts they model, so the evaluation process is defined only from actions associated to some specific concepts.

This design is inspired from the double dispatch of the visitor design pattern that enables extensibility: new kinds of evaluations can be defined and new types of operands can be addressed at will. In this solution, the double inheritance of the original design pattern is replaced by a single template specialization with evaluation and operand types as parameters. Concept-based template specialization allows defining a specific action for any given concept or logical combination of concepts modeled by an operand or operation of an expression.

In the case of operations on vectors and matrices, expression templates provide a significant speed up compared to classic overloading [16]. However, this technique can prevent some compiler optimization due to an aliasing problem [4, 8], or can be revealed inefficient on specific operations where temporary objects are necessary (like matrices multiplication [9]). Concepts can help to address these issues by bringing more semantics to operators (like [6] that assigns algebraic properties to types using concepts) and using a concept-based visitor to adapt the evaluation according to the nature of the operators.

Expression templates based on concepts have been used to define an EDSL for linear programming. With only a few template specializations, an evaluation process to verify, build and simplify linear expressions and constraints has been designed. This experiment shows that concepts offer possibilities of controlling template specialization that enforce its reliability and selectivity. Concept-based specialization should be considered to replace inheritance when dynamic binding is not necessary, as in the case of parsing the static structure of an expression.

References

[1] Bruno Bachelet. Logical Operations on Concepts in the C4TS++ Library. Technical report, LIMOS, Université Blaise Pascal, Clermont-Ferrand, France, 2012.

[2] Bruno Bachelet, Antoine Mahul, and Loïc Yon. Template Metaprogramming Techniques for Concept-Based Specialization. In *Scientific Programming*, volume 21, pages 43–61. IOS Press, 2013.

[3] Bruno Bachelet and Loïc Yon. Schrödinger Effect of Templates. Technical report, LIMOS, Université Blaise Pascal, Clermont-Ferrand, France, 2015.

[4] Federico Bassetti, Kei Davis, and Dan Quinlan. C++ Expression Templates Performance Issues in Scientific Computing. In *Parallel Processing Symposium*, pages 635–639, 1998.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Peter Gottschling and Andrew Lumsdaine. Integrating Semantics and Compilation: Using C++ Concepts to Develop Robust and Efficient Reusable Libraries. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 67–75, 2008.

[7] Douglas Gregor, Bjarne Stroustrup, Jeremy Siek, and James Widman. Proposed Wording for Concepts (Revision 3). Technical report, N2421=07-0281, ISO/IEC JTC 1, 2007.

[8] Jochen Härdtlein, Alexander Linke, and Christoph Pflaum. Fast Expression Templates: Object-Oriented High Performance Computing. In *Lecture Notes in Computer Science*, volume 3515, pages 1055–1063. Springer-Verlag, 2005.

[9] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. Expression Templates Revisited: a Performance Analysis of Current Methodologies. In *SIAM Journal on Scientific Computing*, volume 34-2, pages C42–C69, 2012.

[10] Brian McNamara and Yannis Smaragdakis. Static Interfaces in C++. In *First Workshop on C++ Template Programming*, 2000.

[11] Jeremy G. Siek and Andrew Lumsdaine. Concept Checking: Binding Parametric Polymorphism in C++. In *First Workshop on C++ Template Programming*, 2000.

[12] Andrew Sutton. Working Draft, C++ Extensions for Concepts. Technical report, N4361, ISO/IEC JTC 1, 2015.

[13] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. Concepts Lite: Constraining Templates with Predicates. Technical report, N3580, ISO/IEC JTC 1, 2013.

[14] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: the Complete Guide*. Addison-Wesley, 2003.

[15] Todd L. Veldhuizen. Expression Templates. In *C++ Gems*, pages 475–487. SIGS Books, 1996.

[16] Todd L. Veldhuizen. Arrays in Blitz++. In *Lecture Notes in Computer Science*, volume 1505, pages 223–230. Springer-Verlag, 1998.