Annales Mathematicae et Informaticae
***52** (2020) pp. 5–18*
DOI: https://doi.org/10.33039/ami.2020.12.003
URL: https://ami.uni-eszterhazy.hu

A static analysis method for safe comparison functors in C++*

Bence Babati[a], Norbert Pataki[b]

[a]Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
babati@caesar.elte.hu

[b]ELTE Eötvös Loránd University, Budapest, Hungary
Faculty of Informatics, 3in Research Group, Martonvásár, Hungary
patakino@elte.hu

Submitted: March 17, 2020
Accepted: December 12, 2020
Published online: December 17, 2020

Abstract

The C++ Standard Template Library (STL) is the most well-known and widely used library that is based on the generic programming paradigm. STL takes advantage of C++ templates, so it is an extensible, effective and flexible system. Professional C++ programs cannot miss the usage of the STL because it increases quality, maintainability, understandability and efficacy of the code.

However, the usage of C++ STL does not guarantee perfect, error-free code. Contrarily, incorrect application of the library may introduce new types of problems. Unfortunately, there is still a large number of properties that are tested neither at compilation-time nor at run-time. It is not surprising that in implementations of C++ programs so many STL-related bugs may occur.

It is clearly seen that the compilation validation is not enough to exclude STL-related bugs. For instance, the mathematical properties of user-defined sorting parameters are not validated at compilation phase nor at run-time. Contravention of the strict weak ordering property results in weird behavior

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

that is hard to debug. In this paper, we argue for a static analysis tool which finds erroneous implementation of functors regarding the mathematical properties. The primary goal is to support Continuous Integration pipelines, using this tool during development to overcome debugging efforts.

Keywords: C++, static analysis, STL, generic programming, functor

MSC: 68N19 Other programming techniques

1. Introduction

The C++ Standard Template Library (STL) is a widely-used, handy library based on the generic programming paradigm [2]. On one hand, the library provides convenient, suitable containers (e.g. `list`) and algorithms (e.g. `find`) that make easier stock-in-trade [19]. On the other hand, STL introduces many new kinds of bugs which are hard to detect and fix, such as invalid iterators, weird effect of the `remove` algorithm and writing uninitialized memory via `copy` algorithm, etc. [16]

STL provides four standard sorted associative containers, these are `set`, `map`, `multiset` and `multimap` [8]. These containers are able to work together with user-defined orders via functor types [21]. In this case, the user-defined functor has to implement strict weak ordering, but this property is not validated neither at compilation time nor at runtime [15]. If someone uses a functor which does not fulfill the strict weak ordering rules, the container becomes inconsistent because same values are not considered to be equal [14]. Let us consider the following code:

```
struct Comp
{
  bool operator()( int a, int b ) const
  {
    return a >= b;
  }
};
// ...
std::set<int, Comp> s;
s.insert( 3 );
s.insert( 3 );
std::cout << s.size();
  // Prints 2 that is weird because same value inserted twice
  // into the set. Correctly, 1 should be printed.
std::cout << s.count( 3 ); // prints 0 in spite of it is contained
```

This phenonmenon is weird, the root cause is hard to find. Compilers should emit error (or warning at least) diagnostics, but the problem is not detected at all. Strict weak order property should be an *axiom* according to modern generic constraint approach in C++. However, these axioms are not validated by the compiler [22]. Therefore, our aim is to develop a tool based on static analysis that detects problematic functors.

This tool is based on a recently popular software, called Clang. Clang is a standard compliant C/C++/Objective-C compiler, furthermore, it provides a static analyzer, as well. It is open source and based on the LLVM compiler infrastructure. It is mainly developed by the community, there are many contributors, also it is supported by big companies as well [3].

The Clang architecture is well designed and modular which makes it possible to use it as a library [17]. The users can use the end products, like Clang as a compiler or build their own tools on top of its libraries. It provides an API for third-parties to use its internal structures and analyze the source code in a high-level way. Its libraries provide a wide scale of features related to compilation and analysis, for example tokenizer or AST visitor. Many useful static analysis tools have been developed based on Clang (e.g. [1, 4, 10]). Clang's another significant advantage is the evolving approach regarding the C++ standards, so users do not need to take care of parsing of newly introduced language elements and can focus on their actual goal. That makes Clang powerful and very popular recently.

The rest of this paper is organized as follows: the related work is discussed in Section 2, the technical details of our Clang-based solution are presented in Section 3 and decision logic is explained in Section 4. Our approach is evaluated and results are shown in Section 5. Finally, the paper is concluded in Section 6.

2. Related Work

A comprehensive description of STL-related bugs can be found in [14] including the ordering functor types' mathematical properties, as well. However, many problems have been presented, but no tool support was proposed to avoid the erroneous situations. Compilation time validation of the STL typically uses two different approaches: template metaprogramming (e.g. [18]) and static analysis (e.g. [4, 9]). These methods do not help to find the problematic ordering functor types in C++ source, the functors' statefulness is analyzed exclusively [10]. Model checking of STL containers also misses the validation of user-defined comparisons [6].

On the other hand, C++ functors are analyzed previously, a limited, lightweight, runtime approach has been developed [15]. This approach has runtime overhead and does not deal with comprehensive evaluation.

Another direction in functors' usage is a transparent version of the functor templates [12]. The paper presents a refactoring tool which makes the usage of functors safer, but this tool does not deal with the mathematical properties.

The *constraints* and *concepts* [22] have been included officially in the *C++20* standard version. These let the users to define compile time expectations on the template parameters. For example, it can be checked pragmatically that a given `T` template parameter type has `operator()` member function or not. However, the beforehand presented STL-related issue is more complex, it requires to check the implementation of the given functions as well.

3. Our Approach

3.1. Technical Background

The previously depicted theoretical problem may appear sometimes. However, the compiler cannot warn about it at all. In order to detect this kind of problem, a brand new tool has been developed. Its purpose is to find misuses of ordered associative containers related to the given issue. Many faulty functor classes can be caught in suspicious context, although, the tool has limitations which are described at the end of this section.

The implementation uses Clang's libraries and framework to analyze the C++ source code. It takes advantage of Clang's architecture including the built-in abstract syntax tree (AST) and its visitors. AST is comprehensively used in our tool to extract information from the source code.

3.2. High-level Overview

This section presents a high-level overview and describes how our tool works in a nutshell [5]. As it was mentioned above, it works on the source code itself and it does not require to execute the binary.

That means, it can only rely on compile time information which are given in the source code. The original compiler arguments are very essential regarding the reproducible compilation process. These arguments or flags may affect the whole compilation process, for instance preprocessor macros often depend on the compilation arguments.

In general, let us see what is the idea behind the analysis and how the workflow looks like. This solid outline will highlight the main points of the analysis and how it is performed to gather the necessary information from the source code.

The main problem is related to the associative containers and the regarding user-defined ordering functors. At the beginning, every instantiation of associated containers has to be found which uses a custom functor for comparing objects. The functor classes only can be identified at usage places, because the instantiated assiative container is the evidence of the given functor must meet certain requirements. The beforehand found instantiations each has a functor whose type is a suspect of misusage.

These marked types are analysed in the next step. The tool retrieves the type of comparison functor and tries to find the proper `operator()` for the given usage. Two cases are possible, the definition is not available, for instance it is defined in another translation unit, it will be skipped. This case is rare because most of comparisons have short implementation, so they are typically inline methods in the class. Another case, when the definition of candidate `operator()` is available, it can be analyzed in order to extract the expressions which are used to compare two objects. From one function, multiple expressions can be collected, for example the return value depends on a condition. The following code snippet presents this case:

```
bool ExampleComp::operator()( int lhs, int rhs ) const
{
  if ( lhs > 0 && rhs > 0 )
  {
    return lhs < rhs;
  }
  else
  {
    return lhs * 2 <= rhs + 1;
  }
}
```

These collected expressions are evaluated later in order to decide whether they meet the requirement of strict weak ordering rules. The details of the proposed analysis method can be seen below.

3.2.1. Analyzing AST

In our tool, Clang libraries are in-use to parse the source code and build internal structures. Clang performs every low level action (tokenizing, parsing, etc.) that lets us to concentrate on our aim by defining a higher level analysis based on the built structures.

The main and worth to mention data structure of them is the abstract syntax tree, AST. It represents the source code in an abstract way, contains all the data about the parsed source files. In Clang, it is a little bit more than a syntax tree, because it contains some semantic information as well.

To collect data from abstract syntax trees, they can be visited by AST visitors. Custom AST visitors need to be implemented in order to use the Clang hierarchy and AST visitor interface. AST visitors can extract the relevant information from the AST and capture any kind of context within the AST, for example, all function declarations can be visited.

The proposed tool is mostly built on AST visitors. These visitors can be used to find container instantiations, types, member functions, expressions and many other source-based constructs. More precisely three different kinds of visitors have been declared. Each has different tasks on different part of the AST. These visitors work together and built on each other.

The following paragraphs detail these AST visitors and the presented order is the same as the order of processing. That means in the analysis logic, the visitor which finds associate container instantiations is used before the visitor which parses the body of member functions.

Usage finder visitor The original issue can occur only when someone uses `std::map`, `std::multimap`, `std::set` or `std::multiset` with custom comparison objects. The first task is to find template instantiations of previously listed

types and inspect them in order to find those which are using custom comparison types other than the default `std::less`.

Although, `std::less` can be specialized for used defined types, in this case the written comparator is user-defined and it should be analysed as well. Other special case, when the default `std::less` is provided without any specialization, in this case the `operator<` is called on the objects. The custom object comparison can sneak into without using custom functors. From the analysis point of view, the only difference is that the `operator<` function should be analysed instead of the `operator()` of the provided functor. However, it is not covered in this paper, focusing only on user-defined functors.

When an instantiation meets the given criteria, it should be analyzed because it can be erroneous, for example `SpecialKeyCmp` class is used here:

```
std::map<SpecialKey, int, SpecialKeyCmp> m;
```

After these usage places are located, the classes of the used functors need to be checked. For this, it is necessary to find the definition of the used functor type and the matching `operator()` member function for the given usage. When the definition of `operator()` is available in this translation unit, it can be used to furthermore processing, but it is done by next visitor.

Function body parser The next AST visitor is responsible for parsing the function implementation. Its input is the function definition in the AST, the usage finder visitor passes the `operator()` member function definitions to this visitor.

The visitor's purpose is to extract one or more expressions from the function body which can be used to compare objects. This kind of visitor can locate and capture every logical or comparison expression which can affect the return value. The outcome is a list of expressions which can define the return value of the given function. The visitor needs to process the job backward, because the root expression which defines the return value, can be identified only at the end of each execution path. These end points are the `return`-statements in the function body.

However, it is not adequate to process only them. It can happen that someone declares a local variable or calls a function to evaluate an expression. This visitor needs to handle variable declarations and assignments, when an expression is bounded to a name which is used in `return`-statement. The names are replaced by the bounded expressions in the `return`-statement.

Nevertheless it tracks function calls which can modify variables or their return values appear in the expressions. In case of function calls, the function body is parsed with another object of this visitor to get the relevant expressions.

An important point here is to manage the currently valid conditions on the given execution path. It is necessary, because the conditions can affect the return value, in some case, they define the comparison implicitly. For example, without analysing the conditions, the following functor cannot be judged well, however, it definitely breaks the strict weak ordering rule.

```
bool CustomComp::operator()( int lhs, int rhs ) const
{
  if ( lhs < rhs )
  {
    return false;
  }
  else
  {
    return true;
  }
}
```

In addition to all of this, they need to be performed recursively in order to dissolve an expression as much as possible at compile time. For instance, when a function calls another one which affects the return value in some way, it is necessary to inspect that function and substitute it with extracted elemental expressions.

This visitor deals with the following code context:

```
bool CustomComp::inRange( int value ) const
{
  return value < 42;
}

bool CustomComp::operator()( int lhs, int rhs ) const
{
  const bool tmp = lhs > 0 && rhs > 0;
  return tmp && inRange( lhs ) && inRange( rhs ) && lhs < rhs;
}
```

In this example, the expression which actually will be evaluated at each `operator()` function call is: `lhs > 0 && rhs > 0 && lhs < 42 && rhs < 42 && lhs < rhs`.

Expression parser This is the lowest level visitor in this implementation. This parser works on a very small part of the AST, the beforehand located expressions are visited by it. Its purpose is parsing the given expressions and convert them to an internal data structure. The advantage of this data structure is that, it is far simpler than Clang's AST and contains only the relevant information.

The internal data structure is a graph which represents logical and comparison expressions. The nodes are typically operators and variables but more constructs are supported. The edges are logical relations between nodes, for example, the `operator<` has two other nodes which are the left and right hand side operands of expression.

The visitor handles binary operators, unary operators, literals, variables and so on. It walks over on that small part of AST and tranforms nodes to proper internal data structures. At the end of the visiting of Clang's AST, the result is a

graph which is identical to the original one without excess. For example, the graph belongs to the `a > 0 && b > 0` code snippet is depicted in Figure 1.

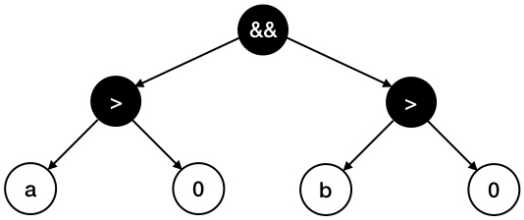

Figure 1: Internal data structure

With this step, the AST processing is mostly done. A list of expressions is extracted for each instantiation which needs to be analyzed later, however, before performing the concrete analysis, some small transformations need to be applied on them. These transformations are detailed in the next subsection.

3.2.2. Transformations

After processing of Clang's AST, an internal graph structure is created for each expression at each functor usage place. They are identical to the original expressions, although most of time they are not that complex. To reduce this complexity, some modifications need to be applied on them. After the transformations, the expressions will be equivalent with the original one, but simpler.

They target to eliminate the obvious complications and keep the expressions plain. There are several well-known replacement rules related to mathematical logic [7]:

- De Morgan's laws: `!(X || Y) -> !X && !Y`
- Double negation: `!!X -> X`
- Tautology: `X && X -> X`

Besides that more transformations can be applied at compile time which comes from the programming language behavior:

- Short-circuit binary operators: at logical `and` and `or`, when the first operand is evaluated, it may define the result of the whole expression, e.g.: `true || (X < 0) -> true`
- Constant evaluation: comparisons may be evaluated at compile time, e.g.: `0 < 42 -> true`

Using these replacement rules, the original expression can be transformed into a new expression which contains less boilerplate. For example, the expression `(x`

`< y) || (0 != 0)` can be converted into `x < y`. The `0 != 0` is not relevant from the analysis point of view since it is always `false` and the outcome of the original expression does not depend on it.

These transformations are applied on each expression when it is possible. This approach results in a new, simplified expression which can be analyzed with more confidence. These newly created expressions will be used later in order to decide the correctness of functors.

3.2.3. Output format

After finding a custom functor suspicious, the tool emits a warning like the compiler does, but it refers to the type that can be seen in the source, not the underlying one [14]. It uses Clang's diagnostic framework to report issues, so they look like a compiler warning at the line of data structure usage, e.g. instantiation of `std::map`.

```
main.cpp:44:10: warning: Strict weak ordering is not fulfilled
                         by comparison type
    std::set<int, Comp> s;
```

4. Decision Logic

The analysis can be executed on cleaned expressions that are prepared to be analyzed whether they meet the requirement of strict weak ordering rules.

Let A be an arbitrary set and relation $R \subseteq A \times A$. It is a strict weak ordering if the following properties are met[20]:

- Asymmetry: $\forall a, b \in A : aRb \Rightarrow \neg(bRa)$.
- Irreflexivity: $\forall a \in A : \neg aRa$.
- Transitivity: $\forall a, b, c \in A : aRb \wedge bRc \Rightarrow aRc$.

On one hand, this analysis is pragmatic and conservative, therefore it minimizes the false positive warnings which is an essential property in static analysis tools, but on the other hand, the tool is not a theorem prover.

The decision logic takes advantage of the previously presented visitors. The pseudocode of the decision logic can be seen in Figure 2, the entry point is the `DecisionLogic` procedure. We omit the proper type information but the informal description helps to comprehend the proposed solution. In this procedure, the first attribute to check whether the comparison uses both arguments because a regular binary relation is required. We use the `ParseNumberOfUtilizedParams` function that is straightforward, therefore we not detailed in Figure 2. If the comparison does not utilize any of its argument, we emit a warning by calling `EmitWarning` that is not detailed in the pseudocode, but presented in Section 3.2.3. However, the functor's `operator()` must have two parameters due to the compilation model of C++ but parameter can be unused [18].

```
procedure CheckExpression(<simplified structure of> expression)
    operator ← ParseOperator(expression)

    if operator is operator== ∨ operator is operator!= then
        EmitWarning
    end if

    if operator is operator<= ∨ operator is operator>= then
        EmitWarning
    end if
end procedure
procedure CheckLiteral(functor)
    literal, expression ← ParseLiteralCondition(functor)

    if ¬(Evaluate(literal)) then
        expression ← ¬(expression)
    end if
    CheckExpression(expression)
end procedure
procedure DecisionLogic(functor)
    params ← ParseNumberOfUtilizedParams(functor)
    if params ≠ 2 then
        EmitWarning
    else
        entity ← ParseReturnEntityType(functor)
        if entity is expression then
            CheckExpression(expression)
        end if
        if entity is literal then
            CheckLiteral(functor)
        end if
        if entity is variable then
            value, success ← ParseVariableValue(functor)

            if success then
                expression ← ParseCondition(functor)

                if ¬ Evaluate(value) then
                    expression ← ¬(expression)
                end if
                CheckExpression(expression)
            end if
        end if
    end if
end procedure
```

Figure 2: Pseudocode for the Decision Logic

If both arguments take part in the comparison, we query what kind of result is specified in the `return`-statement. The potential kinds are expressions, literals (e.g. `true`, or `0`), variables but every kind may depend on function calls that we process by inlining them on the level of AST. However, we do not highlight this fact in Figure 2.

The parameter of the decision logic is the AST representation of the analyzed functor. When we produce the cleaned, simplified expression that we take advantage of transformation steps presented in Section 3.2.2. If this transformed expression contains one of the following operators `<=`, `>=`, `==` or `!=`, we emit a warning, otherwise we consider the comparison meets the requirement conservatively. We query the applied operator with `ParseOperator` method in Figure 2.

When the returned element in the `return`-statement is a literal and the comparison utilizes both parameters the result must depend on a condition. As Section 3 presented, this condition is retrieved by our visitors and the condition is negated when the literal is false or converted to false with the `Evaluate` function. We also showed previously if there are multiple conditional statements, we process all these conditions in the `ParseCondition` function that is not detailed in Figure 2. In case of returned literal is considered to be true by the `Evaluate` method, the condition remains untouched. This condition contains operator to compare the arguments, so we evaluate this processed condition just like the expression previously.

In case of variable is returned, we call the `ParseVariableValue` procedure to recognize its value if we are able to specify it. This recognized value can be used as a literal and evaluate the comparison just like the previous case. We do not emit warning, if the value of the variable cannot be determined. Of course, this can cause false negative cases during the analysis, but it is not a typical use-case.

Briefly, our tool also emits a warning when it detects that the arguments are compared with `operator==` or `operator!=`. If an ordering relation is defined as a C++ comparison functor in an erroneous way, the asymmetry and transitivity requirements are still met. The problematic property is the irreflexivity, therefore our tool focuses on the validation of this requirement that is the most common misuse regarding functors [14]. The possible comparison operators are `<`, `>`, `<=`, `>=`. Although the operators `<` and `>` are considered right, they cannot cause issues regarding to the given problem. The rest of them may cause issues, since the equality is included in all of them, thus we emit warning in these cases.

We also take into consideration whether the arguments are compared with constant values, but they are compared to each other with `<=` or `>=`, therefore this essential expression of functor is incorrect: `lhs > 0 && rhs > 0 && lhs <= rhs`.

5. Limitations and Evaluation

The tool has some limitations, which one should bear in mind. First of them comes from Clang's nature, it handles translation units separately, so if the `operator()` is defined in a different source file (`.cc`) where the container is instantiated with the corresponding functor class, the tool cannot find the operator's definition due

to Clang's limitation [11]. In this case, the given functor will not be analysed.

Another issue is related to compile time behavior, no runtime information is available for the analysis; also if a very tricky comparison expression is written, likely the functor cannot be decided if it is compliant or not.

During the development of the tool, some handmade test cases have been implemented. They are good to cover all the corner cases in theory, however, it would be good to see how the proposed tool performs on real-world projects.

Since the effect of this issue is very well-marked and serious, they usually are eliminated during the development or testing phase of real products.

Nonetheless, in order to ascertain the quality of our approach and solution, the tool was tested and evaluated on well-known open source projects. The user-defined comparison functor usage with associative containers is not used very often, so a limited number of projects could be checked unfortunately. However, even comprehensive profiling does not measure the functors' usage [13].

The methodology of testing was that the tool reported that a functor is being analyzed then the result of the analysis is checked. Each functor which was reportedly analyzed is inspected manually, as well. That makes it possible to verify the result of the tool.

In this testing, four different functors are analyzed from three different projects listed below:

- Flatbuffers - `https://github.com/google/flatbuffers/`
- Thrift - `https://github.com/apache/thrift/`
- Orc - `https://github.com/apache/orc`

All of the analyzed functors are used with `std::map` container. None of them was reported as suspicious by our tool and the manual verification proved the results' correctness. Despite of the limitations of the tool, every functor's properties are evaluated correctly. The limitations do not affect the usage of tool in the source code of real-world applications. The tool does not emit false positive reports at all, so it can be used safely in quality assurance regularly.

6. Conclusion

C++ STL is a widely-used library that is based on the generic programming paradigm. The usage of the library increases the code quality and comprehensibility, however, the incorrect usage of library may result in new kind of errors.

This paper has presented a weird error related to the C++ Standard Template Library that is related to sorted associated containers. The ordering can be customized via functor class, but it should implement strict weak ordering. However, this property is not validated at all. If a functor does not meet this requirement, the container becomes inconsistent.

So in order to detect this kind of defects in the source code, a new approach has been proposed. We have developed a tool for this method. The proposed solution

analyzes source code that means the execution of the program is not required. It is a Clang-based tool that takes advantage of Clang's libraries and framework. Our tool was tested on manually prepared test cases and it was evaluated on open source projects to prove that it works perfectly with real-world applications.

The tool did not find any questionable functor, however, it confirms our tool validity and the fact that is not a very often issue in released projects. Although it does not report unnecessary false positive alarms, so it can be a handy tool in the development process and Continuous Integration servers for quick feedback, as well.

References

[1] M. Arroyo, F. Chiotta, F. Bavera: *An user configurable Clang Static Analyzer taint checker*, in: 2016 35th International Conference of the Chilean Computer Science Society (SCCC), Oct. 2016, pp. 1–12,
doi: 10.1109/SCCC.2016.7835996.

[2] M. H. Austern: *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley, 1999, isbn: 0-201-30956-4.

[3] B. Babati, G. Horváth, N. Pataki, A. Páter-Részeg: *On the Validated Usage of the C++ Standard Template Library*, in: Proceedings of the 9th Balkan Conference on Informatics, BCI'19, Sofia, Bulgaria: ACM, 2019, 23:1–23:8, isbn: 978-1-4503-7193-3,
doi: 10.1145/3351556.3351570,
url: http://doi.acm.org/10.1145/3351556.3351570.

[4] B. Babati, N. Pataki: *Analysis of Include Dependencies in C++ Source Code*, in: Communication Papers of the 2017 Federated Conference on Computer Science and Information Systems, ed. by M. Ganzha, L. Maciaszek, M. Paprzycki, vol. 13, Annals of Computer Science and Information Systems, PTI, 2017, pp. 149–156,
doi: 10.15439/2017F358,
url: http://dx.doi.org/10.15439/2017F358.

[5] B. Babati, N. Pataki: *Static analysis of functors' mathematical properties in C++ source code*, AIP Conference Proceedings 2116.1 (2019), p. 350002,
doi: 10.1063/1.5114355, eprint: https://aip.scitation.org/doi/pdf/10.1063/1.5114355,
url: https://aip.scitation.org/doi/abs/10.1063/1.5114355.

[6] N. Blanc, A. Groce, D. Kroening: *Verifying C++ with STL Containers via Predicate Abstraction*, in: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 521–524, isbn: 9781595938824,
doi: 10.1145/1321631.1321724,
url: https://doi.org/10.1145/1321631.1321724.

[7] A. Church: *Introduction to mathematical logic*, Princeton University Press, 1996, isbn: 978-0691029061.

[8] D. Das, M. Valluri, M. Wong, C. Cambly: *Speeding up STL Set/Map Usage in C++ Applications*, in: Performance Evaluation: Metrics, Models and Benchmarks, ed. by S. Kounev, I. Gorton, K. Sachs, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 314–321, isbn: 978-3-540-69814-2.

[9] D. Gregor, S. Schupp: *STLlint: lifting static checking from languages to libraries*, Software: Practice and Experience 36.3 (2006), pp. 225–254,
doi: 10.1002/spe.683, eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.683,
url: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.683.

[10] G. Horváth, N. Pataki: *Clang matchers for verified usage of the C++ Standard Template Library*, Annales Mathematicae et Informaticae 44 (2015), pp. 99–109,
URL: http://ami.ektf.hu/uploads/papers/finalpdf/AMI_44_from99to109.pdf.

[11] G. Horváth, N. Pataki: *Source Language Representation of Function Summaries in Static Analysis*, in: Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS '16, Rome, Italy: ACM, 2016, 6:1–6:9, ISBN: 978-1-4503-4837-9,
DOI: 10.1145/3012408.3012414,
URL: http://doi.acm.org/10.1145/3012408.3012414.

[12] G. Horváth, N. Pataki: *Transparent functors for the C++ Standard Template Library*, in: Proceedings of the 11th Joint Conference on Mathematics and Computer Science, ed. by E. Vatai, CEUR-WS, 2016, pp. 96–101.

[13] P. Jungblut, R. Kowalewski, K. Fürlinger: *Source-to-Source Instrumentation for Profiling Runtime Behavior of C++ Containers*, in: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), June 2018, pp. 948–953,
DOI: 10.1109/HPCC/SmartCity/DSS.2018.00157.

[14] S. Meyers: *Effective STL*, Addison-Wesley, 2001, ISBN: 0-201-74962-9.

[15] N. Pataki: *Advanced Functor Framework for C++ Standard Template Library*, Studia Universitatis Babeş-Bolyai Informatica LVI (2011), pp. 99–113.

[16] N. Pataki: *C++ Standard Template Library by safe functors*, in: Proc. of 8th Joint Conference on Mathematics and Computer Science, MaCS, 2010, pp. 363–374.

[17] N. Pataki, T. Cséri, Z. Szűgyi: *Task-specific style verification*, AIP Conference Proceedings 1479.1 (2012), pp. 490–493,
DOI: 10.1063/1.4756173, eprint: https://aip.scitation.org/doi/pdf/10.1063/1.4756173,
URL: https://aip.scitation.org/doi/abs/10.1063/1.4756173.

[18] N. Pataki, Z. Porkoláb: *Extension of Iterator Traits in the C++ Standard Template Library*, in: Proceedings of the Federated Conference on Computer Science and Information Systems, ed. by M. Ganzha, L. Maciaszek, M. Paprzycki, Szczecin, Poland: IEEE Computer Society Press, 2011, pp. 911–914.

[19] N. Pataki, Z. Szűgyi, G. Dévai: *Measuring the Overhead of C++ Standard Template Library Safe Variants*, Electronic Notes in Theoretical Computer Science 264.5 (2011), Proceedings of the Second Workshop on Generative Technologies (WGT) 2010, pp. 71–83, ISSN: 1571-0661,
DOI: https://doi.org/10.1016/j.entcs.2011.06.005,
URL: http://www.sciencedirect.com/science/article/pii/S1571066111000764.

[20] F. Roberts, B. Tesman: *Applied combinatorics*, CRC Press, 2009.

[21] B. Stroustrup: *The C++ Programming Language (special edition)*, Addison-Wesley, 2000, ISBN: 0-201-70073-5.

[22] A. Sutton, B. Stroustrup: *Design of Concept Libraries for C++*, in: Software Language Engineering, ed. by A. Sloane, U. Assmann, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 97–118, ISBN: 978-3-642-28830-2.