EVALUATION OF PREDICATES IN THE C++ STANDARD TEMPLATE LIBRARY

Bence Babati and Norbert Pataki
(Budapest, Hungary)

Communicated by András Benczúr
(Received May 1, 2021; accepted July 17, 2021)

Abstract. C++ is widely used language nowadays, it provides many high-level constructs. Its standard library tries to minimize classical programming errors, however, it introduces new ones. One of the known STL issues is related to algorithm predicates. Predicates make the algorithms customizable and they sometime store states for their task. However, these stored variables can introduce unexpected behavior. This issue can be detected in many ways, but standard compilers cannot detect this problem. We developed multiple techniques which are presented in this paper and compared to each other. These approaches cover both static analysis and dynamic analysis methods.

1. Introduction

C++ provides a standardized library with common functionalities which is called Standard Template Library (STL). The library is part of the C++ standard [1]. It is a general software library with many well-known functionalities for common problems. However, the library is based on the generic programming paradigm, therefore it is a header-only library, most of the functions and classes are template. It aims to reduce the classical programming errors, however, it introduces others [2].

The library can be divided into different parts based on the provided functionalities. One of them is the containers, for example, `std::vector`. The

Key words and phrases: C++, static analysis, dynamic analysis, STL algorithm, predicate.
2010 Mathematics Subject Classification: 68N15.
1998 CR Categories and Descriptors: D.2.4.
https://doi.org/10.71352/ac.52.057

most commonly used containers are available, for instance, lists, arrays, sets and maps as well. All of them are templates that mean one can store any kind of elements in them. However, these elements must fulfill some conditions, for example, `std::set` requires that the stored element must have `operator<` defined.

Other important part is the algorithms. Many well-known algorithms are already defined, so the users should not reimplement them all the times. For example, sorting algorithms (`std::sort`, `std::stable_sort`) or minimum/-maximum element search algorithms (`std::min_element`, `std::max_element`) are already implemented. It is very useful from the users point of view that one can simply use an algorithm which is surely implemented well and can focus on higher level problems. These algorithms are templates as well because this construct makes it possible to work on any kind of containers which fulfills the conditions.

These containers and algorithms are implemented in a generic way and intended for general usage [3]. That means any of the containers can work with any algorithms almost. Also user defined containers can work with STL algorithms as well. There minor limitations are specific for given algorithms, for example, `std::sort` can work on containers which provide random access to elements, so it cannot work on `std::list`.

The relation between the containers and algorithms is provided through the iterators. The iterators are also part the STL, their purpose is to provide access to the elements of a container without knowing anything specific about the container [16]. Their interface is very similar to pointers, for example, they have `operator->` and `operator*` for element access. The iterator abstraction hides the underlying container and abstracts the element access and puts behind an interface. This abstraction connects the algorithms and containers. The iterators can be categorized into multiple categories based on how they behave with the underlying container. For instance, random access iterators must provide `operator[]` to access the elements by index.

The containers can decide on their own which kind of iterator they provide. For example, `std::list` provides bidirectional iterator, so the elements cannot be accessed by indexing. This constraint restricts the set of algorithms which can work with this container.

These algorithms additionally to the iterators may accept an extra function parameter which is a predicate. The predicate is function-like object which can be used as a function. It can be a functor, function pointer or a lambda. These predicates are used to make a modification or a decision on an element. For example, the `std::find_if` algorithm takes a predicate parameter, this predicate gets an element and returns true if it is the searched one.

The predicates are very useful for customizing the algorithms and can execute very complex tasks also. For their task, they need to store values in some cases, for example, removing the element which is equal to the stored value. It is completely valid use case. These stored values behave like a state sometimes, which is read and/or written in the `operator()`. However, it can cause issues when the return value depends on this state.

Let us see an example.

```
struct LastEven {
  bool operator()(const int value) {
    const bool result = lastEven;
    lastEven = value % 2 == 0;
    return result;
  }

  bool lastEven = false;
};

int main() {
  std::vector<int> v { 1, 2, 3, 5 };
  std::remove_if(std::begin(v),std::end(v),LastEven{});
  ...
}
```

In this example, it is intended to remove every element which is after an even number. The expected behavior is to remove value 3. If this predicate is instantiated once and called on every element once, the result is the expected. However, it is possible that the previous preconditions do not met. The `remove_if` may use another function which takes the predicate by value so the original state is cleaned or the algorithm creates multiple copies of the original predicate and use all of them.

This means that the predicates cannot rely on states surely, because there are cases when it does not work as expected. The result depends on the STL implementation as well. If it does not copy and use multiple copies of the predicate then states safely can be used within predicates.

These STL related issues is a well-marked error, it is usually fixed in early development or testing phase, however, it is hard to debug. There are a few tools to detect various STL related issues [7], however, in this paper we concentrate on stateful predicates.

To detect this issue, we present our own different approaches, including both static and dynamic analysis techniques. However, the main focus is on the comparison of the presented methods. The methods are examined based on different testcases. Through these examples, the differences of the presented methods can be demonstrated. In Section 2, we discuss related work and later, in the further sections, these methods are detailed and the comparison is revealed after. Finally, this paper concludes in Section 6.

2. Related work

The comprehensive description of the incorrect usage of STL are presented [2]. The semantically verified usage of STL is substantial [7]. Compile-time and runtime approaches are available, for instance, a runtime validation method for iterator invalidation is presented [19]. However, in most of the cases, the compile time methods are preferred. These methods typically use static analysis or template metaprogramming. For example, a template metaprogramming approach for detecting intricate instantions is realized [17]. A compile-time solution that takes advantage of STL's iterator traits type for safe coping and searching algorithms is proposed [18]. Template metaprograms are not so sophisticated, therefore static analysis methods are preferred. STLlint is the first software artifact which aims at the validation of STL usage with static analysis [9]. However, its support and availability is cancelled. On the other hand, a modern, Clang-based approach is presented [10]. In this solution, predicates are validated whether they are stateful in a naive way which results in unnecessary false positive findings. Moreover, other runtime solutions are presented, as well. Method based on aspect-oriented programming is known [7]. Debugger-based STL implementation-dependent runtime validation is also available [7].

3. Static analysis methods

Static analysis is a kind of software analysis, the main idea is to analyze a software without executing it, checking only the source of the software. The source can be source code or byte code, both techniques are applied. The advantage of this way is that, it is not necessary to recreate an execution environment including the dependencies and data. It can be executed in a build environment or it is not even necessary.

Many static analysis methods are evolved recently, from simple pattern matching to abstract interpretation [8] or symbolic execution [12]. These techniques could be different in implementation complexity and produced result as well.

3.1. Abstract syntax tree visiting approach

The earlier presented STL usage issue may appear sometimes, however, it has a very characteristic effect that means it cannot be hidden for a long time. One of the applied techniques to detect this problem is Abstract Syntax Tree (AST) visiting.

It is a known static analysis method which makes it possible to analyze source code on a high level representation with syntactical and semantical information [11].

A new static analysis tool has been developed to recognize this issue. This tool is using Clang for the analysis. Clang is an open source compiler suite for C/C++/Objective-C which is based on LLVM infrastructure [4]. It provides also a static analyzer. A lot of supporters contribute in its development, including large firms.

The modular architecture makes Clang very unique. In addition to the delivered compiler and static analyzer, Clang provides many APIs and libraries which expose internal functions for public use. These provided APIs and libraries can be used to build various tools by third party developers [6]. It is future proof in these days to use Clang as library in order to make analyzer tools. It simplifies the third party developers business because they do not need to take care of low level source code processing tasks and can focus primarily on the high level tasks. For example, the C++ standard changes should not necessarily be handled, just in case of the developed tool's purpose requires it.

The developed tool takes advantage of Clang by using the parsing, tokenizing and other low level tasks. Including the abstract syntax tree building as well. This tool relies on the AST and AST visiting interface of Clang.

The abstract syntax tree is a high level representation of the source code. It contains syntactical information mostly, however, Clang's AST has some semantic information too. It is a tree structure which defines the relations of different language elements from the sources. For example, the `main` function is an AST node and its content is the subtree of its AST node.

Based on the abstract syntax tree, the developed tool can analyze the original source code. It visits the AST in different ways to find suspicious patterns and validate them. The first target is to find STL algorithm usages. These points can be the starting points for the analysis.

When an STL usage is hit, the tool must validate the given usage. There are two main conditions to check:

- Algorithm predicate copying must be detected, because it is the main reason behind this problem. Also additionally, it is necessary to use multiple copies of the predicate in the algorithm. This constraint is difficult to properly detect only from the source code, however, it is possible with a high certainty.
- Predicate implementation should be checked as well, because state like variables are necessary to be watched. These variables contribute to the faulty behavior of the predicate when they are read and written in the function call operator. Also it is necessary that the return value relies on these variables too.

By fulfilling these conditions, an STL algorithm usage place can be marked as problematic. After the analysis, the tool can emit warnings, which indicate where the original algorithm call happened.

However, these conditions are hard to analyze properly in practice. During the implementation there were many challenges which maybe obvious in the description, however, difficult to formalize them. First of them is to find algorithm usages. How can we know that a given function is an algorithm? Multiple answers are possible.

The most simple answer is that its definition comes from `algorithm` header file. However, the STL headers can be fragmented, which means that the algorithm definition placed in an internal header [5].

They can be known by name because the Standard defines them very precisely. It works, but should be maintained continuously.

A heuristic can be defined to detect an algorithm, which is not hundred percent sure, but in practice no false positive and negative cases have been seen. The function should be placed in `std` namespace, it must be a function template, the last template parameter is a type and has a function parameter with this type. The advantage of this method, that it is not necessary to maintain. The tool is using this method to detect algorithms.

Second challenge is to detect whether an algorithm copies a predicate. A predicate copying can happen through parameter passing or explicitly in the function body. A copy of an object can be made by calling copy constructor or assignment operator. These functions appear in the AST and can be tracked.

However, the hard part is to decide whether the newly copied object is used or not. It is necessary to judge the function, if it is actively using multiple predicate objects. The usage is a function call operator call on the specific objects. However, tracking these calls to given objects is difficult in the AST. For example, function calls must be followed from the algorithms as well.

This last condition makes the analysis implementation dependent. It is useful if we define two subtypes within this method:

- Implementation independent approach, where only the user code is under analysis.
- Implementation dependent approach, when the source of the STL is checked as well as it was described above.

The last topic is analyzing the predicate object whether it is using variables. These variables can be members or global variables, also can be local variables in case of lambda function with capture. A variable is used when both read and write happen on it. Both reads and writes appear in the AST, however, for different types different functions should be watched. For example, there are more possibilities for a user defined type to be modified, than a default type. On the other hand, function calls must be followed from here as well because it is possible that a helper function handles the variable reading and writing.

3.2. Type-traits-based approach

Standard type traits are available since C++11. Type traits process user-defined types at compilation time and one can analyze the basic characteristics of own classes [17]. This approach supports compile-time solution, however, to realize how many copies are constructed, we take advantage of type traits at runtime. This solution makes the results exact.

First, we crate a template to detect if a type is stateless:

```
template <class T>
struct is_stateless
{
  static const bool value =
    std::is_class<T>::value &&
    std::is_empty<T>::value;
};
```

Some utilities to make the usage easier:

```
#define PREDICATE_COPY_CHECK(PRED)
    copy_safe<PRED>::reset()

struct too_many_copies
{
  // information for exception handling
};
```

The heart of this approach is the `copy_safe` class template:

```
template <class Pred>
class copy_safe
{

  static int cnt;

public:

  copy_safe() = default;

  copy_safe( const copy_safe<Pred>& )
  {
    ++cnt;
    if ( !is_stateless<Pred>::value && cnt > 1 )
    {
      throw too_many_copies();
    }
  }

  static void reset()
  {
    cnt = 0;
  }

};
```

The `copy_safe` class template keeps track how many copies are constructed and throws exception in case of a stateful predicate is copied more multiple times. The template parameter is the type of the predicate. This approach can be used in a non-intrusive way:

```
class Pred: public copy_safe<Pred>
{

  // ...

  bool operator()( int a ) const
  {
    // ...
  }

};
```

When an object of type `Pred` is copied, the copy constructor of `copy_safe` is invoked, as well. This approach follows the actual copy constructor calls inside the STL, therefore it perfectly works with every STL implementation.

4. Runtime approach

Other big group of analysis methods is dynamic analysis. It is a runtime approach, where the compiled binary is executed in some way. It is usually instrumented at compile time or at runtime. Many methods have been evolved also. Advantage of this runtime approach is that real values are seen during execution, therefore the covered software parts can be evaluated more safely. However, this runtime environment and data for covering the more code branches in the execution must be prepared.

4.1. Aspect-oriented programming approach

The aspect-oriented programming is a programming paradigm [13], however, it can be used as a runtime analysis technique [15]. The main idea is to add extra code on top of the original source code. It requires no modification in the analyzed source code. Only an extra layer is added which is independent from the original code. The extra code parts could be written using a language extension or another language. Most of the time it requires an additional step in the compilation process too.

Many aspect-oriented paradigm supporting libraries and tools exist for different programming languages. One of them is AspectC++ which is designed for C++ programs [14]. It lets one waive C++ codes with custom code parts. It extends the C++ language with new elements in order to enable the waiving. The aspect related parts are using this language extension, however, within these parts the standard C++ code can be written.

Let us see an example. In this example, the `hello_world` function calls are caught and after the function calls, it inserts a function which will be executed. In this inserted code, only a logging line is printed to the standard output. The `pointcut` defines a pattern where the aspect matches, after the match is hit, the body of the advice will be executed.

```
aspect Example {
  pointcut hwpc() = "void hello_world()";

  advice execution(hello_world()) : after() {
    std::cout << "hello_world() function called"
              << std::endl;
  }
};
```

The AspectC++ is using Clang for doing its task. This approach uses a two phased compilation. In the first phase, it takes the aspect definitions and the original source. It parses the original source code with Clang and commit AST level changes according to the given aspect. When this transformation is ready, it generates C++ code from the modified AST. In this second phase, this generated source can be compiled with a standard compliant compiler to get the binary. The resulted binary already contains the changes defined by the aspect.

This technique can be used in many cases, for example for analysis too. Many aspects were developed to find STL related issues [7]. Including an aspect which is able to find the stateful predicate usage issue. The implementation is fully different from other techniques. The idea behind validation is to analyze the behavior and not the content of the predicate. Analyzing the content of the predicate is much harder at runtime and the behavior defines the predicate. Let us see a part of the defined aspect:

```
aspect StatefulPredicate {
  pointcut removeifpc() =
       "% std::remove_if<%s, %s>(...)";

  advice call(removeifpc()) : before() {
    ...
    PredT origPred = *tjp->arg<2>();
    std::vector<ResultT> firstResults =
      applyPredicate(*tjp->arg<0>(), *tjp->arg<1>(),
                     origPred);

    PredT copiedPred = origPred;
    std::vector<ResultT> secondResults =
      applyPredicate(*tjp->arg<0>(), *tjp->arg<1>(),
                     copiedPred);

    if (firstResults != secondResults) {
      std::cout << "std::remove_if is used "
                << "with stateful predicate at "
                << tjp->filename() << ":"
                << tjp->line() << std::endl;
    }
  }
};
```

First of all, a `pointcut` is defined which matches on given STL algorithms, `std::remove_if` in this example. After an algorithm call is found, the proposed approach extracts the iterators and the predicate object from the call. The behavior analysis uses these parameters, so it iterates over the given iterator range and calls the predicate on every element. The original predicate will be copied after and this newly created predicate is used to test the elements of the iterator range again. From both runs, the results of each predicate is stored, so if the predicate depends on some kind state, the results of the two runs are not the same. In case it happens, a warning can be emitted at runtime. The drawback of this method is that only the executed parts are covered, but this limitation comes from the runtime behavior.

5. Evaluation

Multiple analysis methods have been presented to catch the same STL related issue [7]. These methods approach otherwise, trying to grab the problems from different angles. In this section, these methods are compared to each other.

There are multiple properties for the comparison: outcome, sophistication, performance.

The performance cannot be judged reliably, since there are compile time and runtime methods as well.

Another point is sophistication, which depends on the behavior of the analysis method. The advantage of runtime methods is that they can see real data and make decisions based on them. However, the static analysis methods see only the source code and can make assumptions based on them. Static analysis methods do not require runtime environment which is an advantage in most of the cases.

The outcome of each analysis methods can be measured through comparison test cases which cover the general use cases and the corner cases as well. These examples are the base points of the comparison. Both false positive and false negative cases are covered. They decide how many issues can be detected with each analysis method. The testcases are defined in this section, they help to explore the differentiations of the solutions.

5.1. Comparison cases

5.1.1. Functor with member variable

A classic example, which is using a member to track the algorithm state. For example, counting how many times the predicate was called and behave based on this counter.

```
struct CallCounter {
  bool operator() (const int x) {
    return ++times == 42;
  }

  std::size_t times = 0;
};
```

Using this predicate, may lead erroneous behavior, however, this fact is not fully clear at this point.

5.1.2. Algorithm implementation dependency

One step ahead is to improve the first example and take the algorithm implementation into account. Most of the cases, the previously sampled predicate just works fine. That is because of the algorithm implementation.

Each algorithm implementation is not strictly regulated by the standard, so the implementations may vary between STL libraries. Let us see two implementations of the `std::remove_if`. Both of them do the same, however, they work a bit differently.

```
template<typename FwdIt, typename UnaryPred>
FwdIt remove_if(FwdIt begin, FwdIt end, UnaryPred p) {
  begin = find_if(begin, end, p);
  if (begin == end) {
    return begin;
  } else {
    FwdIt next = begin;
    return remove_copy_if(++next, end, begin, p);
  }
}
```

The second implementation, which is not using helper functions:

```
template<class FwdIt, class UnaryPred>
FwdIt remove_if(FwdIt begin, FwdIt end, UnaryPred p) {
  for(FwdIt it = begin; it != end; ++it) {
    if (!p(*it)) {
      *begin = *it;
      ++begin;
    }
  }
  return begin;
}
```

As it can be seen, the behavior of each function is the same, however, the outcome may be different. Let us combine them with the previous predicate which returns true for the third element. With the first `std::remove_if` implementation, it removes the third and sixth elements as well if the `find_if` takes the predicate by value. While with the second one, only the third element will be removed.

5.1.3. Using global variables

Another interesting example, when a predicate does not explicitly define a member or a locally used variable to store information. Despite they use globally available variables, which can be used exactly in the same way as a member variable.

```
int GlobalX = 0;

struct Foobar {
  bool operator() (const int x) const {
    return ++GlobalX == 42;
  }
};
```

In this example, the function call counter is tracked in global variable. It is the same issue as it is seen in the first example, however, it is much harder to catch. This is mostly a theoretical case, it would appear in real code base very rarely.

5.1.4. Lambda functions with state

Lambda functions can be used as algorithm predicates since C++11. A lambda can have state by capturing a variable by value. In this case, the behavior is the same as in case of functors with member variable. When it is copied, the state within is copied as well.

```
int main() {
  int count = 0;
  auto cfn = [count] (const int x)
                { return ++count == 42 };
  ...
}
```

5.1.5. Read-only and unused state

The most frequently used pattern is using constant variables in predicates. However, it does not cause any issue since it is not a state. Almost the same behavior, when only writing a variable, however, the value is not used related to the return value or at all.

```
struct Found {
  explicit Found(const int x) : x(x) { }

  bool operator() (const int y) const {
    return x == y;
  }

  int x = 0;
};
```

These are negative cases which do not cause any issue, but they are potential false positive hits.

5.2. Evaluation with real-world examples

So far, we have not evaluated our methods with real-world examples because these are quite different and take advantage of STL implementation in order to reduce the false positive findings. Runtime approaches hard to drive in real-world examples because it should cover all potential execution or an analysis is required how to use the examined software to reach specific points in the source code that takes too much effort. Moreover, released software artifacts typically do not specify which version of which STL implementation take advantage of. Our findings could not be reproduced with a different library implementation and we cannot extract the library information from the source or the compiled code. However, our previous experiences show that the proposed solutions can be applied with real-world examples [5]. Additionally, a Clang-based STL usage validation is utilized by industrial partner [10].

5.3. Comparison summary

From the previously presented cases and methods, the comparison can be seen in the Table 1. In this table, TT stands for type traits and refers to the corresponding method, AST is for abstract syntax tree visiting based approach and the AOP is for aspect-oriented programming.

Validation	TT	AST	AOP
Functor with member	+	+	+
STL implementation dependency	-	+	-
Global variables	+	+	+
Lambdas	-	+	+
Read-only state	-	+	+
Unused state	-	+	+

Table 1. Comparison of approaches by testcases

Each method has advantages and drawbacks and approaching the issue differently.

The AST based approach can cover all the presented cases in general, however, there could corner cases where the evaluation may fail, for example a function call cannot be properly followed in the AST may lead false negative cases.

The aspect oriented approach can insert validation code parts to the code, but it requires the execution of the program. Most of the cases can properly be analyzed at runtime, however, it is hard to insert validation code and properly follow calls in the STL implementation.

The type trait based approach require to user interaction to derive the predicates from the `copy_safe` class which can track the safe usage at runtime. Consequently, only the user defined code parts can be validated. It means that the STL dependency cannot be verified and the pattern of the state usage cannot be validated. Therefore the lambdas cannot be checked too, because they cannot be derived from the given `copy_safe` class.

6. Conclusion

C++ STL is most well-known and widely-used library based on the generic programming paradigm. In this paper, an STL usage issue is presented which is related to the algorithm predicates. It is covered with detailed description and examples. We developed and presented analysis techniques which aim at the detection of this exact issue. Our approaches involve both static and dynamic analysis methods.

The depicted methods are detailed on how they can detect the algorithm predicate issue. Since this issue is very complex, the analysis methods grasp the essence of this issue from different sides. Both advantages and drawbacks are covered for every analysis method by collecting and analyzing the most frequent testcases of this issue. At the end, a comparison is demonstrated including multiple viewpoints.

References

[1] **Stroustrup, B.,** *The C++ Programming Language (special edition)*, Addison-Wesley, 2000.

[2] **Meyers, S.,** *Effective STL*, Addison-Wesley, 2001.

[3] **Stepanov, A.A. and D.E. Rose,** *From Mathematics to Generic Programming*, Addison-Wesley Professional, 2014.

[4] **Lattner, C.,** *LLVM and Clang: Next generation compiler technology*, The BSD conference. Vol. **5**, 2008.

[5] **Babati, B. and N. Pataki,** Analysis of include dependencies in C++ source code, in: *Annals of Computer Science and Information Systems, Vol 13*, pp. 149–156, 2017.

[6] **Babati, B., G. Horváth, V. Májer and N. Pataki,** Static analysis toolset with Clang, in: *Proceedings of the 10th International Conference on Applied Informatics*, 2017, pp. 23–29.

[7] **Babati, B., G. Horváth, N. Pataki and A. Páter-Részeg,** On the validated usage of the C++ Standard Template Library, in: *Proc. of the 9th Balkan Conference on Informatics*, 2019, 23(1)–23(8).

[8] **Cousot, P.,** Abstract interpretation, *ACM Computing Surveys* (*CSUR*), **28(2)** (1996), 324–328.

[9] **Gregor, D. and S. Schupp,** STLlint: lifting static checking from languages to libraries, *Software: Practive and Experience*, **36(3)** 2006, 225–254.

[10] **Horváth, G. and N. Pataki,** Clang matchers for verified usage of the C++ Standard Template Library, *Ann. Math. Inform.*, **44** 2015, 99–109.

[11] **Jones, J.,** Abstract syntax tree implementation idioms, in: *Proceedings of the 10th conference on pattern languages of programs* (*plop2003*), 2003, p. 26.

[12] **King, J.C.,** Symbolic execution and program testing, *Communications of the ACM*, **19** (1976), 385–394.

[13] **Kiczales, G.,** Aspect-oriented programming, European conference on object-oriented programming, Springer, Berlin, Heidelberg, 1997.

[14] **Spinczyk, O. and D. Lohmann,** The design and implementation of AspectC++, *Knowledge-Based Systems*, **20(7)** (2007), 636–651.

[15] **Shin, H., Y. Endoh and Y. Kataoka,** Arve: aspect-oriented runtime verification environment, *International Workshop on Runtime Verification*, 2007, 87–96.

[16] **Pataki, N.,** Safe iterator framework for the C++ Standard Template Library, *Acta Electrotechnica et Informatica*, **12(1)** (2012), 17–24.

[17] **Pataki, N.,** Compile-time advances of the C++ Standard Template Library, *Annales Univ. Sci. Budapest., Sect. Comp.*, **36** (2012), 341–352.

[18] **Pataki, N. and Z. Porkoláb,** Extension of iterator traits in the C++ Standard Template Library, in: *Federated Conference on Computer Science and Information Systems*, 2011, 911–914.

[19] **Pataki, N., Z. Szűgyi and G. Dévai,** Measuring the overhead of C++ Standard Template Library Safe Variants, *Electron. Notes Theor. Comput. Sci.*, **264(5)** (2011), 71–83.

B. Babati and N. Pataki
Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
Budapest
Hungary
`babati@caesar.elte.hu`
`patakino@elte.hu`