STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LX**, Number 1, 2015

TOWARDS SAFER PROGRAMMING LANGUAGE CONSTRUCTS

ÁRON BARÁTH AND ZOLTÁN PORKOLÁB

Abstract. Most of the current programming languages inherit their syntax and semantics from technology of the 20th century. Due to the backward compatibility, these properties are still unchanged, however newer technologies require different language constructs and different semantics. Instead of redefining the programming language, the developers enhance the language with new library functions, or they add some – occasionally ambiguous – elements to the syntax. Some languages provide very loose syntax, which is harmful, because it leads to critical errors. In other case the interleaving "normal" code and exception handling code can obfuscate the developer itself and the subsequent developers.

This paper highlights several aspects of language elements such as basic and potentially unsafe elements of the syntax, control flow constructs, elements used in const-correctness, type-system, elements of multiparadigm programming – generative and functional –, capabilities of embedding a DSL, parallelism support, and taking account of branch prediction. These aspects determine the usablity, safety and learnability of a language. This paper also gives recommendation for a new and safe experimental programming language.

1. Introduction

Current mainstream object-oriented languages contain several problematic constructs which potentially lead to critical errors. Most of the errors came from the loose syntax or not proper semantics. In our work and research we saw a lot of harmful codes, and we intended to give recommendations to extend the coding style or to design a new programming language in order to avoid malicious constructs.

Received by the editors: May 1, 2014.

2010 *Mathematics Subject Classification.* 68N15.

1998 *CR Categories and Descriptors.* D.3.3 [**Software**]: Programming Languages – *Language Constructs and Features*.

Key words and phrases. programming languages, compilers, syntactical vulnerabilities, semantical vulnerabilities, compiler techniques, safe language.

The programming languages have been created in order to reduce assembly errors and to increase the abstraction level. The early languages (e.g. COBOL, LISP, ALGOL, FORTRAN [18, 11]) introduced higher abstraction level, but in this early age, languages were imperfect – they contain many serious issues in syntax and semantics. After the fundamental paradigms invented, the structure of the programming languages started to lose their major gaps in syntax and semantics. And the C programming langauge has beed released [9] in this era.

Nowadays, most of the programming langauges contain syntactic and semantic legacy from the early programming langauges, however, the basic concept is obsolete. The modern processors and computers are designed to execute multiple tasks at the same time, but the early languages did not support that. Because of the huge existing code bases, the developers of the programming languages – for example the C++ – keep the language to be backward compatible. This decision can be debated, and the backward compatibility precludes important security and safety changes in the language.

Our research is based on the alpha version of our experimental programming language, which is currently contains intention to be a syntactically and semantically safe language. The research aims to improve our language, furthermore identify and highlight the vulnerabilities of the current programming languages. This paper includes but not limited to the presented finding.

This paper is organized as follows: In Section 2 we present critical syntactical errors in current mainstream programming languages, we give recommendations about const-correntness and we analyze the loop constructs with examples. In Section 3 we describe main semantical features, we highlight vulnerabilities of the type-system (like infinite loop caused by improper comparisons or unwanted implicit casts) and exceptions (injecting the rarely used exception handler codes into the normal control-thread can mislead the developers and makes the code less understandable), furthermore, we discuss the necessity of the multiparadigm languages (generative and functional language elements, implementing uncommong data structures). Section 4 is about the extended features for support present technologies, e.g. embedding a domain-specific language or writing multithread programs – while staying only at the language features. In Section 5 we overview code generation, like the usage of built-in branch prediction in the programming languages and in the modern CPUs. In Section 6 we shortly brief our experimental programming language. Finally, in Section 7 we describe our development plans. Our paper concludes in Section 8.

2. Syntactical elements

The syntax of a programming language is its face, and it can be judged by the programmers. Because the programmers are the *users* of the programming languages, it is important to keep the syntax as clear, pure and intelligible as possible. Adding loose elements to the syntax can be very harmful, because the reason of some critical errors can be attributable to the loose syntax.

In the following we discuss some syntactical vulnerabilities and improvements.

2.1. **Permissive syntax.** Recently, articles appeared on the Internet about a mistake in a source code belongs to Apple Inc. [23] The reason of the error is a mistakenly duplicated line containing only the `goto fail;` statement – that is why this error called as *goto fail*. This error resulted a serious security leak.

Two problems can be identified in the source code in Figure 1. First, in the *then* branch of the *if* statement we can see only a single statement instead of nested inside a block statement, i.e. came from the permissive syntax of the C programming language. Second, a proper coding style should avoid the use of unconditional jump statements.

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail; // duplicated line here
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
```

Figure 1. The affected lines of the *goto fail* error.

Using block statements in every branches of the *if*, *for* and *while* statements is not without precedent – many coding styles require the block statement for reason. Furthermore, adding empty lines makes the code more legible. Although, the block delimiters and empty lines stretch the code, they provide more benefits, e.g. making it more readible.

Applying the recommendations, there is an improved version of the code as can be seen in Figure 2. At this point, the duplicated `goto fail;` statement does not affect the code at all. Even the duplicated line will be ignored by the compiler, so there is no overhead at runtime. Furthermore, the duplicated line is a dead code, so the compiler can detect it, and the programmer can get diagnostics message.

Note that, this feature exists in other languages influenced by the C programming language: C++ [20], C# [1], and Java [4]. In C++, the *goto* statement is also available. However, C# and Java do not have *goto* statement

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
{
  goto fail;
  goto fail; // duplicated line here
}

if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
```

FIGURE 2. Transformed version of the error handling.

(as its classical meaning), but the block statement is not required – this can lead to similar error like *goto fail*. Using the `break;` statement is attainable a similer problem.

Because the loose part of the syntax caused the *goto fail* error, we recommend mandatory block statements in the coding style. Even better if the programming language does not allow standalone statement after *if*, *for* and *while*. The syntax of our experimental language requires block statement after the concerned control statements. This is a nice way to avoid errors which are undetectable by the compiler.

2.2. **Constant variables.** In declaration of variables or parameters the programmer should take into account of constness. In C++, the const-correctness deals with which variables or objects are mutable and which are immutable [6]. This is a compile-time construct, so it has to be noted in the source code. The most common way to express the constness is to tag the type with a modifier. In C++, is frequent to pass function arguments as *const* or as *const-reference*.

C++ uses the `const` qualifier to make a variable or an object *read-only*. C# and Java provides a similar mechanism to express constness. Using the `final` qualifier in Java makes the variable unassignable, thus the variable must be initialized in the declaration. Note that, the content of a `final` variable is still modifiable. In C#, the `readonly` keyword has the same effect as the `final` in Java. Furthermore, C# introduced the `const` qualifier, which has an effect similar to the `#define` in C++. However, this feature has some shortages, for example parameters can not be marked as `const`.

As we have seen, in current mainstream languages the default behaviour is the mutable. Notable counter-example the C++11 lambda functions [8], where the default is the `const` mode. The lambda is a syntactical sugar, and compiled as embedded structs. By default, the generated functor is a constant member function. This is also an example, where a new syntax element do not have to be backward compatible with earier elements. Since it is a new language construction in C++, it uses a safer approach.

The Version 2 of the D programming language supports two different aspects of constness. A variable can be mutable and `immutable`, the view of a mutable variable can be `const`, and the view of an immutable variable can be `const` as can be seen in Figure 3 [21].

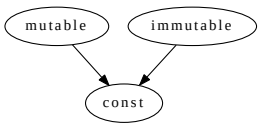

FIGURE 3. constness relationships in D programming language.

Unfortunately keeping the const-correctness of the code is not simple. In C++, almost all code can be compiled without using any `const` qualifier. And based on the fact that programmers do not want to write more than necessary, the `const` will not be placed everywhere where needed. An other problem is to teach and to account const-correctness.

Let approach this problem from a totally different aspect: do not require `const` qualifier for immutable variables, but require the `mutable` qualifier for variables and arguments which can be modified. Using this technique, the compiler will enfore to keep the const-correctness of the code. At this point, the programmer can forget only the `mutable` qualifier, but the compiler will calling to account the missing qualifier. Therefore, the code will be more safer thanks to the inverted psychology – programmers can not neglect mandatory keywords.

Parameters	236
By value	87
Constant	105
Mutable	44

FIGURE 4. Function parameter analysis of TinyXML.

Might arise the question that more or less keywords must be used in the code. An analysis about TinyXML 4 can be seen in Figure 4 – the rate of the constant function arguments is about a half to the total number of arguments. However, the analysis showed, the parameters passed by value is relevant. This implies that, the number of mutable arguments (*total number of arguments - constants - by value*) is less than half of constant arguments. Based on this analysis, using our suggestions half of qualifiers are required, and the compiler will check the const-correctness (via requiring the `mutable` qualifier instead of `const`).

2.3. **Control flow.** Sometimes the programmer needs to understand someone else's code. It is a great help if the artifacts belong together are collected in one group. The most common issue in this topic is the `while` and the `for` loops. Almost every programming language supports at least two different loop construction, and the `while` and the `for` loops are sufficiently widespread.

The key problem with a common `while` loop is the loop-variable. Three activities can be identified: the initializing-, the role in the loop-condition-, and the stepping of the loop-variable. These pieces can be scattered into the statements of the `while` loop. Obviously, the code is obfuscated, and this is only the programmer's fault.

This implies the problem: how to force the programmer to use the proper loop construction. Take an example from a C++ code: given a variable of `std::vector<int>` [12, 2, 15], print its the content to the standard output. Strive to use only the language features. The first construction in Figure 5 can easily mislead the programmer who would like to understand the code.

```
int idx = 0;
while(idx<vec.size())
{
  std::cout << vec[idx] << std::endl;
  ++idx;
}
```

FIGURE 5. A poor implementation of iterating an array.

Therefore, gather the related pieces together: use the `for` statement and place the three parts next to each other. The new construction in Figure 6 is completely understandable, but not the best solution, because the code uses the indexing operator, thus the code is less portable.

```
for(unsigned int idx=0;idx<vec.size();++idx)
{
  std::cout << vec[idx] << std::endl;
}
```

FIGURE 6. An other implementation of iterating an array.

The modified code satisfy the portability requirement, because the it can be generalized easily. Unfortunately, the code became larger and more complex as can be seen in Figure 7. This piece of code uses the the iterators of `std::vector<int>`. This is not so comfortable, and must be replaced with a better construction.

```
for(std::vector<int>::const_iterator
  it=vec.begin(),end=vec.end(); it!=end; ++it)
{
  std::cout << *it << std::endl;
}
```

FIGURE 7. Using the iterators of `std::vector<int>`.

The final solution came from the idea that why the programmer should write a simple iteration again and again every time, while the construction is quite the same. Also, writing more code means more possibility to make a mistake. The related code can be generated automatically by the compiler. The C++11 standard [20] introduced the *foreach* mechanism, and its usage and the final version of the code can be found in Figure 8. The *foreach* mechanism is available in other languages such as Java, C#, go, python, etc.

```
for(int x : vec)
{
  std::cout << x << std::endl;
}
```

FIGURE 8. A safe implementation of iterating an array.

This is quite good and safe solution, because the programmer wrote what container has to be iterated, and not how the container has to be iterated. We recommend that to use the *foreach* mechanism where possible, and avoid the fragile and misunderstandable constructions.

2.4. **Summary.** As can be seen above, restricting the syntax can improve the code. It is important to declare additional rules in the coding conventions, for example "use braces in all control statements". In case of designing a new language, it is a good start to require explicit compound statements to avoid malicious contructs. Also, the need of the `goto` statement must be considered – high-level languages rarely support it.

Based on the presumption, that programmers are lazy, the usage of the reversed philosophy of the constness improves the quality as can be seen above. Even if the changed syntax may result longer code, the quality improvements are more beneficial.

3. Semantical elements

The following section describes the used type-systems in programming languages. The aim is to uncover the vulnerabilities of the constructs, and give a safer alternative implementation. We used these alternatives in our experimental programming language.

3.1. **Type-system.** There are two different approach of the type-systems. The first is the completely dynamic type-system, where the variables do not have types in the source code. This type-system is commonly used in script languages and in some functional languages, for example in Erlang [17]. The dynamic type-system is a new opportunity for freedom, but understanding the code (i.e. static analysis) is much harder. The second is the static type-system, which is widespread in imperative- and object-oriented programming languages. The static type-system is used to express the intended type for every variable. It helps to understand the code. This also implies an additional check by the compiler. In the following we assume that, the programming language uses static type-system.

Even the static type-system is fairly safe, problems are hidden inside. For example, the implicit cast mechanism is sometimes useful, but it can result very harmful situations, such as infinite loops, and hardly followable or unpredictable code behaviour.

In the C++ language we can define constructors with one parameter. Without the `explicit` keywords these contructors can be called implicitly as conversion. Thus, the compiler can create, for example, complex object from a single integer. Moreover, C++ classes can also define conversion operators, and classes can be converted to primitive types without explicit cast. These features can mislead the programmer who wants to understand the source.

```
// [1]
std::vector<int> vec;
for(int i=0;i<vec.size();++i) { /* ... */ }
// [2]
unsigned int x = 0xffffffffu;
if(-1 == x) { /* ... */ }
```

Figure 9. Comparison of signed and unsigned integers (C++).

In C++, semantically wrong comparisons can be written, but the compiler indicates only a *warning* – however, there are major semantical errors. The code in Figure 9 contains two semantical errors. The first piece is a common error: comparing a signed and an unsigned integer can outcome bad result.

The compiler will show a warning for this part. The second piece is worse than the first one, because checking the equality of an unsigned an signed value is meaningless. The problem is that, the representation of the largest unsigned value (`0xffffffffu`) and the *minus one* are the same. The compiler accepts the code, the condition of the *if* statement is true, but no warnings and no errors are generated.

The origin of these problems is that the signed and unsigned intergers have different domain, although the half of the domains overlap. There are two solutions for this problem. For example, the Java programming language does not introduced unsigned integers. Still the implicit casts can ruin a Java program as well: the code can be seen in Figure 10 is an implicit infinite loop. Another solutions is not to allow comparisons between different types at all – our experimental language uses this solution.

```
// Valid code in Java, C++, and C#
for(char ch='\0';ch<70000;++ch) { /* ... */ }
```

FIGURE 10. Infinite loop caused by implicit cast.

3.2. **Multiparadigm support.** The languages which support multiparadigm programming are more flexible, because the programmer can use the proper paradigm for the specific task. The most conspicuous is the C++11 standard, because of support for generic programming (especially template metaprogramming) [7], object-oriented programming, and functional programming. In C++, the functional programming is observable in the template metaprogramming, and in the lambda expressions.

The highligthed feature is the lambda expressions: introduced as a useful feature, but it can easily go wrong. Basically, the lamba are used to replace unnecessary boiler-plate codes with a compact syntax. Due to these lamba are very sematic, the compiler can generate the affected embedded structs – the result is the same, but with a safer syntax. However, when the programmer writes larger lamba, the understandability of the code is heavily decreasing.

Furthermore, the *union* contruction of the C++ can be connected with the algebraic data types from functional languages. Although, the *union* can be implemented with abstract class and derived classes, it needs a massive amount of code. The task is to store different values, and to handle them as uniform. Staying as close to the language features as possible, we do not take account the `Any` from the *boost* library. There is a huge problem with the *union* construction: it is possible to "cast" a character to a pointer at any time without any compiler feedback. However, the algebraic data types are much safer than the *union*, and it is a closed type. The common *tuple*

type (for example in Erlang) is an open type, the programmer can create a completely new variant of a tuple without modifying the type declaration – but not with the algebraic data types.

Consequently, the algebraic data types could be a powerful feature in C++. This gave the inspiration to us, to implement the algebraic data types in our experimental language.

The generics in Java programming language is a syntactical sugar introduced to help the programmer to write type-safe collections and type-safe classes [5, 4]. Most of the Java programs before JDK 1.5 contained massive amount of basically unnecessary casting, and a `ClassCastException` could be thrown. However, after the program passed the static analisys phase, the compiler omit this type-info from the class file – this itechnique is called *type-erasure*.

3.3. **Exceptions.** As mentioned above, understandibility and clear-looking code is important. Another critical topic is the exception handling. The exception is raised or thrown when an exceptional event happened – for example: program ran out of memory, or an index is out of range. The program must be prepared to handle the exceptions, but these parts of the code may not executed at all. However, some languages, like Java, C#, and C++, allow – and requires – interleaving of "normal" and exception handler code. Thus, the train of thought have breaked many times, and the programmer can follow the code hardly. This effect appears most frequently in Java and C#.

The idea of the solution came from the Eiffel programming language [14]. The Eiffel supports exception handler code only at the end of the function. In the exception handler code the programmer can write the `retry` instruction to restart the execution of the function.

If the exception handler code is placed at the end of the function, the programmer will enforced to write shorter functions – which is good, because large functions are hardly understandable. Although, in Java, is very unusual and uncomfortable to write the exception handler codes only at the end of the function. However, this technique is a nice way to keep the code organized. Our experimental language uses this technique for placing exception handlers in order to keep the understandability of the source code.

3.4. **Summary.** Based on the presumption, that programmers tend to write the possible minimal code, the loose static type-system can be very harmful. The implicit casts and mixed exception handlers can decrease performace and the understandability. Using a strict static type-system and a strict exception handler syntax results a clear code and improves the code quality.

4. Extended features

Nowadays, it is a requirement to a programming language to support special features, for example embedding a domain-specific language, or writing parallel programs. It is important to use only the features provided by the language, and work without any external tool (script for preprocessing, third party tool) or library in order to keep the portability of the code [16].

Moreover, it is certain optimization to use the features provided by the compiler to implement a domain-specific language or a parallel program, because the compiler is used to know the best solutions for the specific platform.

There are many ways to embed a domain-specific language (DSL) into an existent language [19]. We used the type-system and the operator overloading mechanism to create DSLs. The reason was obvious: the compiler can check all of the expressions, and discover any problematic statement – using only the strong static type-system. The usage of the operators is a convenient feature, because the syntax of the created DSLs are similar to the original language.

In a lot of programming languages the support of parallelism is not part of the language. For example, C, C++, and Pascal can handle threads with library functions. An other example is the C# and Java languages, because these provide a `Thread` class, and the programmer can easily write multithreaded algorithms. Not that, the `Thread` class is not considered as a library, because this is the part of the basic runtime library, and no C#, nor Java program can be executed without it.

There are more possible ways to implement the thread model in a virtual machine [10]: all the threads are an operating-system thread, or some of them are mapped to an OS thread, maybe none of them. Our virtual machine uses a dynamic virtual-thread mapping mechanism – run the thread on the first idle OS thread (in the virtual machine's terminology, the OS threads are the "processors"). Due to this approach, creating a new thread inside the virtual machine does not require massive amount of resources.

4.1. **Summary.** Using a multiparadigm or easily extendable language can keep portability of the code, because no external tools are required. Concurrent programming is essential in modern software technology, therefore the language support of multithreading is important. This trend can be observed in the C++ programming language, since the C++11 standard introduced the threads in the standard library.

5. Code generation

The native programs can be accelerated by the features of the CPU, for example the branch prediction or return address prediction. These features are

built-in into a modern processor to speed up the execution. The results can be seen in Figure 11. The related codes aim the branch prediction feature of the compiler and the CPU. The programs compiled without any optimizations (but using built-in branch predition). This is used to enable runtime CPU optimizations.

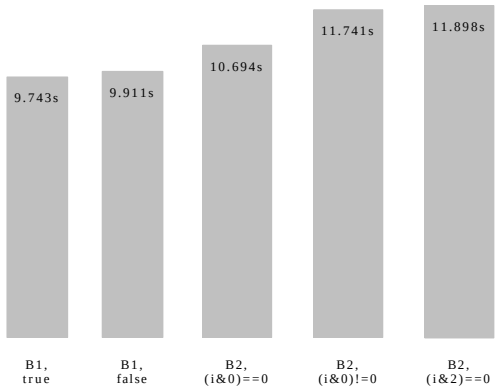

FIGURE 11. Results of the branch prediction test.

```
void sum(bool cond)
{
  int sum1 = 0, sum2 = 0;
  for(unsigned int i=0;i<0xffffffffu;++i)
  {
    if(cond) { ++sum1; }
    else     { ++sum2; }
  }
}
```

FIGURE 12. Code used in B1 test (C++).

Note that, the type and the value of the `sum` variables in codes in Figure 12 and in Figure 13 is not relevant.

However, the codes above after minor changes (upper limit is decreased to `0x7fffffff`) has been used in Java programs too, but the Java compiler and the Java Virtual Machine (JVM) uses massive optimizations. Therefore, there are no difference at runtime with the code in Figure 12. Interesting results came from the code in Figure 13, because when the condition is potentially true (condition is `(i & 0x0)==0`), the Java optimized the whole loop out –

```
void sum()
{
  int sum = 0;
  for(unsigned int i=0;i<0xffffffffu;++i)
  {
    if(<COND>) { ++sum; }
  }
}
```

FIGURE 13. Code used in B2 test (C++).

resulting almost zero time in execution. But, when the condition is not trivial, the execution of program took 3.152 seconds.

It follows that, there are benefits to compile to a virtual machine like JVM, because more runtime optimizations can be applied. Note that, the measurement above is artifical, and may not appear in product code.

5.1. **Summary.** There is several ways of program portability. First, compiling the same code on multiple platforms, and the compiled executable performs the same actions – as the C and C++ languages. Second, compile the source code once, and use the same executable on multiple platforms. This technique requires a virtual machine, or a higher level compiler (JIT). There are arguments in favor of both techniques.

6. WELLTYPE

Our experimental programming language is called *Welltype*[1]. Welltype is a strongly typed imperative programming language, which is based on C and C++ languages, also influenced by Ada and Eiffel languages. The main concept is to minimalize errors caused by typos, type mismatches, unclear control structures, etc. This is ensured via the strict and clear syntax, and the strong static type-system. Some language constructions cause more verbose source code, but the langauge prefers safety over compact code. For example our language handles assignments as statements instead of expressions, like C-style languages to avoid unwanted side effects at assignment.

This programming language is designed to include all recommendation in this paper, and we rendered an example language to show that our findings can be used in practice. The prototype implementation of this programming language is available at `http://baratharon.web.elte.hu/welltype`.

[1]The *Welltype* is an intentionally wrong abbreviation.

6.1. **Permissive syntax.** Our language is invulnerable to the *goto fail* error, since requires braces in every control statement constructions, such as `if`, `while`, `do-while`, and `for`, also there is no `goto` statement in the language. These restrictions are aim the clear and structured programs.

6.2. **Constness.** The Welltype language uses the reversed constness philosophy described earlier. All object parameters are passed as immutable references, but the `mutable` modifier can be placed to make the content of the parameter mutable.

6.3. **Type-system.** In order to keep the type-system clear, our language uses strong static type-system. This implies that, there is no automatic type casting, and implicit object creation. Therefore, all function calls and assignments can be validated by the programmer, since it is based on the static types and the validation is a straightforward algorithm. Every expressions and literals have exactly one type at compile-time, and that type can be converted only explicitly.

6.4. **Multiparadigm support.** We are working on to introduce the algebraic data types and the generics in our programming language.

6.5. **Exceptions.** Our programming language supports an Eiffel-like exception handling: the exception handler can be placed at the end of the function, therefore the exception handler will not interleave with the "normal" logic.

6.6. **Extended features.** The preceding version of the Welltype language was used to implement embedded domain-specific languages (eDSL). We find out, the implemented operator overloading mechanism is adequate for several features, for example implementing DSLs [3].

7. Future work

Based on the findings described in this paper, we will improve the alpha version of our experimental language. The most important intention is the make the syntax and the sematics as safe as possible, and to detect the most of the errors. We will analyse the modified syntax and semantics of our language to find out the attainable vulnerabilities.

In order to ensure about the recommendations we made, we intended to make a personal survey with programmers and non-programmers as well. The survey will consists of two parts: writing a program individually to test the constructions of the language, and adding a new feature into an existent code to test the comprehensibility of the language. By way of comparison, we will use beside our language C++ or Java.

8. Conclusion

In this paper we inspected the safety of different aspects in programming language, for example syntax, semantics, extended features and code generation. We discovered that, the originate of many vulnerabilities and harmful elements is the legacy of the 20th century and sometimes the backward compatibility. We have given recommendations to avoid harmful constructs and to keep the safety and understandability of the code.

The analysis started with the syntactical elements, and we can see that, many problem can be traced back to the loose syntax of the language. This is mainly occur in C programming language and other languages with the similar syntax. In other hand, the proper accessibility notations of the objects (variables or parameters) have added value – in C++, means const-correctness. Due to the permissiveness of the compiler, the semantically missing `const` qualifiers will not be reported. However, the reversed approach can eliminate the problem, and provides more compile-time checks.

The next step was the analysis of the semantical elements. The main aspects was the deficiency of the type-system, e.g. in C++, C# and Java, the programmer can easily write infinite loops without any compiler feedback. The cause of this unfortunately problem is the implicit casts. In our experimental language there are no implicit casts to avoid these kind of errors. We showed that, the exception handlign in mainstream object-oriented languages can break the normal execution thread in the source code, causing harder understandability.

We discussed the opportunity and features of embedding a domain-specific language into an existing host language. In our experimental language we prefer to use the type-system and the operator overloading mechanism instead of external, third party tools. Because our language does not support implicit casts, there are no chance to break the DSL's types unintentionally. An other perspective is about writing multithread program with langauge support only. For example, in C and C++ there is no language support to handle threads.

Finally, we tested the basic built-in branch prediction mechanism in the compiler, and runtime impacts by a modern CPU. We experienced that, the compiler assumes that, the condition of the branch is always true – including *if*, *for*, and *while* statements. It implies that, the programmer should take the branch prediction into account.

References

[1] Albahari, J. and Albahari, B.: C# 4.0 in a Nutshell: The Definitive Reference, O'Reilly Media, 2010

[2] Austern, M. H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library, Addison-Wesley, 1998

[3] Baráth, Á., Porkoláb, Z.: Domain-Specific Languages with Custom Operators, To appear in proceedings of *The 9th International Conference on Applied Informatics*, 2014
[4] Bloch, J.: Effective Java (2nd Edition). Prentice Hall PTR, NJ, USA, 2008
[5] Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *OOPSLA '98 Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Pages 183–200, ACM New York, NY, USA, 1998
[6] Cline, M. P., Lomow, G. and Girou, M.: C++ FAQs, Pearson Education, 1998
[7] Czarnecki, K., Eisenecker, U. W.,: Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000
[8] Järvi, J., Freeman, J.: C++ lambda expressions and closures. Science of Computer Programming 75.9 (2010): 762-772.
[9] Kernighan, B. W., and Ritche, D. M.: The C programming language. Vol. 2. Englewood Cliffs: prentice-Hall, 1988
[10] Manson, J., William P., and Sarita V.: The Java memory model. Vol. 40. No. 1. ACM, 2005
[11] Metcalf, M., Ker Reid, J., and Cohen, M.: Fortran 95/2003 Explained. Vol. 416. Oxford: Oxford University Press, 2004
[12] Meyers, S., Effective C++, Third Edition, Addison-Wesley, 2005
[13] Meyers, S. Effective STL - 50 Specific Ways to Improve Your Use of the Standard Template Library, Addison-Wesley, 2001
[14] Meyer, B.: Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 1997
[15] Pataki, N., Szűgyi, Z., Dévai, G.: *C++ Standard Template Library in a Safer Way* , In Proc. of Workshop on Generative Technologies 2010 (WGT 2010), pp. 46–55.
[16] Porkolab, Z., Sinkovics, Á.: Domain-specific Language Integration with Compile-time Parser Generator Library. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, Pages 137–146, ACM New York, NY, USA, 2010
[17] Laurent, S- St.: Introducing Erlang, O'Reilly Media, 2013
[18] Schach, S. R.: Object-oriented and classical software engineering. Vol. 6. New York: McGraw-Hill, 2002
[19] Sinkovics, Á, Porkoláb, Z.: Domain-Specific Language Integration with C++ Template Metaprogramming. Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. IGI Global, 2013. 32-55. Web. 30 Apr. 2014. doi:10.4018/978-1-4666-2092-6.ch002
[20] Stroustrup, B. The C++ Programming Language, 4th Edition, Addison-Wesley, 2013
[21] const(FAQ) - D Programming Language. http://dlang.org/const-faq.html#const
[22] TinyXML, http://www.grinninglizard.com/tinyxml2
[23] Vulnerability Summary for CVE-2014-1266. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266

Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University

E-mail address: {baratharon|gsd}@caesar.elte.hu