Towards safer programming language constructs

Theses of the doctoral dissertation

2018

Áron Baráth

baratharon@caesar.elte.hu

Thesis advisor: **Dr. Zoltán Porkoláb, docent**

Eötvös Loránd University, Faculty of Informatics,

1117 Budapest, Pázmány Péter sétány 1/C

ELTE IK Doctoral School

Doctoral program: Foundations and Methodologies of Informatics

Head of the doctoral school: Prof. Dr. Erzsébet Csuhaj-Varjú

Head of the doctoral program: Prof. Dr. Zoltán Horváth

---

1 Motivation

In 1953, John W. Backus working at IBM submitted a proposal to develop a practical alternative to the then used assembly language for programming the IBM 704 mainframe computer. The first documentation was finalized in late 1956 and half a year later the first operating FORTRAN compiler was in use. Although, the possibility to generate machine code from a high-level programming language was well known, there was a great skepticism of the effectiveness of such a solution. Therefore, FORTRAN targeted to generate code with performance comparable to that hand-coded assembly language. Safe language constructions were neither a goal nor well known at that time.

Modern programming languages are not just about higher abstraction level unlike old languages but aimed to be safer by giving numerous validations. Language evolution is directing toward safer languages. Obviously, a safer language requires more resources to compile in general, but a lot of time can be spared during development as the strong and strict type system saves the programmers from many semantic issues. Nowadays the compilers are fast enough, and most of the programmers will not perceive the overhead of the extra work. However, the user will experience the benefits of a stricter language, because less run-time checks are necessary.

In this thesis we discuss the importance of these safe language constructs. In the document we will present examples from current mainstream languages where either the syntax or the semantics allow constructs leading to possible mistakes. Many of these examples are based on actual bugs happened in major companies. We analyze the root cause of these issues and suggest solutions for the existing mainstream programming languages – mainly for C++. As a proof-of-concept we designed the Welltype language and we implemented as a prototype to prove that the recommendations we made are usable.

2 Syntax

Current mainstream 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 my everyday work and research I saw a lot of harmful codes, and I intend to give recommendations to extend the coding style or to design a new programming language in order to avoid malicious constructs.

A good example to present how important is a proper coding style is the vulnerability introduced into operating system developed by Apple Inc. [11]. The reason of the error is a mistakenly duplicated line containing only the `goto fail;` statement – that is why this error is called as *goto fail*.

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

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

As can be seen on Figure 1 the problem was a duplicated line. This kind of errors can be prevented by applying a proper coding style – for instance always use block statements. An alternative solution is to use a programming language – e.g. Go – that require block statements and reports error if the block statement is missing. Also, this recommendation or syntactical restriction will reveal other erroneous codes that can be seen on Figure 2.

```
if(some_condition); {
  do_something();
}
```

Figure 2: Erroneous *if* statement (extra semicolon breaks the code).

Furthermore, we examined the how the const-correctness affects a language. We argue how convenient to use it: for example in C and C++

languages support the `const` keyword. It is also an argument in what level the programming language should enforce the use of consts. In C and C++, for example, it is possible to write code that compiles with or without warnings, but will eventually crashes due to writing to a read-only memory area. Thus, the const-correctness should be enforced.

We also examine how control-flow constructions affect the code and its comprehension. Some solutions will increase the quality of the code, others will decrease it. This statement is also true for operators. Finally, we discuss the effects of the operator overloading, and how important is to implement domain-specific languages.

We present our solution to the revealed issues we aimed in the Welltype language. In Welltype we implemented the const-correctness mechanism by automatically consider all function arguments as immutable. When the programmer wants to modify an argument, actions must be explicitly taken: the argument must be declared as `mutable`. If the programmer forget to specify the `mutable` keyword in the code, and a modifier operation is performed on the argument, it will result in a compiler error. Enforcing const-correctness with this mechanism is trivial. It is, however, neither easy nor trivial to ensure const-correctness in C or C++.

Moreover, our solution to avoid potential errors derived from scattered code fragments, and misleadingly overloaded operators are part of the Welltype specification. These are long discussed in the thesis.

Thesis 1 I have examined the vulnerabilities related to the syntax of various mainstream programming languages. I have identified (1) the permissive syntax, (2) using variables as mutable memory areas by default, (3) non-expressive control flow, and (4) inconsistent definition of operator overloading as major source of possible software bugs. I suggested fixes and workarounds to the problems above for the current programming languages, especially for C++. I specified the syntax of the Welltype prototype programming language to demonstrate how these problems could be avoided by applying carefully defined syntac-

tical rules.

3 Semantics

There are several approaches to the type-systems. One extreme approach is a completely dynamic type-system, where the source code does not determine the type of an object. Such type-systems are commonly used in script languages and in some functional languages, for example in Erlang [12]. The dynamic type-system is a new opportunity for freedom, but understanding the code (i.e. static analysis) is much harder. The opposite approach 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.

The origin of some problems is that the signed and unsigned integers have different domain, although the half of the domains overlap. That is the reason why it works in most of the cases – but eventually it will break in irregular cases, those are rarely covered by tests. 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 3 is an implicit infinite loop. Another solutions is not to allow comparisons between different types at all – our experimental language uses this solution. Thus, the code snippet in Figure 3 causes a compile error.

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

Figure 3: Infinite loop caused by implicit cast – compile error in Well-type.

As demonstrated earlier the permissive syntax and semantics in C++ [13, 14] can lead to harmful situations. Most of these constructs can be compiled without any compiler warning message.

We introduced a working solution in C++ to eliminate implicit conversions [3]. Our approach uses a wrapper template class, and we can precisely define what operations can be performed on a type. Thus, we can detect unintended type conversions.

The C and C++ languages have value semantics [14, 15]. Objects of C and C++ – either having built-in or user defined types – can exist in the stack, in the static memory or in the heap. In all cases variables identify the raw set of bits of objects without intermediate handlers. When we apply assignment, we copy raw bits by default.

Since C++11 move semantics [13] can avoid unnecessary copy thus may increase run time performance. However, it is hard to implement move operations, and such mistakes lead to hard-to-detect performance penalties. Our method [2] and prototype tool [10] is analyzing such issues to detect unintentional copy operations. The backbone of the solution is the generalized attributes that are introduced in C++11. This mechanism allows to place user-defined attributes to numerous elements, and these attributes can be processed. We introduced our attributes that guide the method to detect copy-instead-of-move defects.

Thesis 2 I investigated the negative consequences related to the permissive type systems, especially the implicit conversions in the current mainstream programming languages. I designed and implemented a wrapper class based solution to solve the problem in C++. I specified an algorithm which detects the possible misuse of C++11 move semantics. I created and tested a prototype tool implementing the algorithm. I specified the semantics of the Welltype language to avoid the mentioned issues and defined the data types with their expected behavior.

4 Compiling, linking, binary compatibility and testing

Modern programming languages support modular development dividing the system into separate translation units and compile them individually. A linker is used then to assemble together these units either statically or dynamically. This process, however, introduces implicit dependencies between the translation units. When one or more units are modified in inconsistent way binary incompatibility occurs and may result in unexpected program behavior. Current mainstream programming languages neither specify what are the binary compatibility rules nor provide tools to check them.

The Welltype language aimed to avoid binary incompatibilities: since the Welltype dynamic loader *recursively validates* all imported elements, it is able to detect incompatibilities. The signature of the functions are validated, including the name of the functions, number and type of the arguments and the returned types, and the `pure` property. Records are also recursively validated, which consists of name of the record, and number, type and name of the fields.

As we seen above, binary compatibility is really an issue in long-term development. The incompatibilities can cause the program to crash or, even worse, miscalculations that break invariants. Thus, programmers must take actions to avoid such incompatibilities. While in the current mainstream programming languages only unofficial conventions can get rid of binary incompatibilities, the Welltype language explicitly and strictly specifies the binary compatibility.

Modern software development have to take testing into account to ensure the reliability of the software product. Test-driven development requires to specify the new features first by writing new test cases, and after implement it to fulfill the test cases. This method provides clear specification for the new features. Moreover, any reported bugs become test cases, and the way of fixing it is the same. Finally, all written test

cases are part of the regression test.

Two major kind of tests are known: black-box and white-box testing [16]. The *black-box* tests are focused on the input and the output: for specific input, the specific output must be provided, and no matters how. The *white-box* tests are dedicated to get as high code coverage as possible by providing different inputs to execute distinct parts of the code.

The unit tests can be handled different ways. The tests can be written by hand as any regular program, and refer to the libraries which are tested. This technique is too inflexible, because numerous additional code is required to make detailed error messages.

The C++11 introduced the `static_assert` mechanism, thus the programmers can write compile-time assertions. Furthermore, the `constexpr` is introduced for optimization purposes, since the compiler is entitled to evaluate all `constexpr` expressions.

Putting the `static_assert` and the `constexpr` together, we can write tests which are evaluated at compile-time. The compile-time tests are the aid for all problems which came from the third-party tools, because all actions are performed by the compiler. So, there is no additional dependencies, and the C++ project is more portable. Using compile-time tests results more reliability, because the source code will not compile if one of the tests fail.

Using this technique, any test cases can be written inside the source code, and the compiler will evaluate them during the compilation.

Thesis 3 I analyzed the problem of the binary compatibility for current mainstream programming languages. I have identified that software bugs can be introduced due to violating the API via binary incompatible components. I specified the rules of binary compatibility for Welltype programming language in order to prevent binary incompatible modules being loaded and linked. I implemented a prototype binary loader application to test the developed method to detect binary incompatibility issues at link time. I presented a possible solution to improve

the unit testing in C++ by executing the unit tests during compilation. The solution relies on new features in C++11.

5 A prototype proof-of-concept language

In the predecessor theses we described possible solutions to numerous issues in current mainstream programming languages. In order to prove that those design changes are viable, we implemented a prototype language that designed according our findings. Our language is an imperative language with additional multi-paradigm language elements. We named this language Welltype, and it is available as an open source project [9].

We found the language usable, because it operates with minimal syntactical and compilation overhead. The language, however, has a rich set of features which makes the language convenient to use. The compiled programs do not suffer additional run time penalty, because the language has a static type-system.

Relevant publications for theses

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Syntax	X	-	-	-	-	X	X	-
Semantics	X	X	X	X	-	X	-	-
Linking	X	-	-	-	X	X	-	X

6 Summary

Current mainstream programming languages suffer numerous safety issues that originally introduced to be convenient and practical, but later turned out these features can be harmful. Although, these programming languages continuously evolving, they usually have a limitation due to backward compatibility. New languages are being created

today to fix issues or introduce new paradigms.

In thesis 1 we describe various software issues related to language syntax based on real industrial experiences. To avoid these problems we suggest strict coding conventions for existing languages and rigorous syntactical rules for new languages. Rules include to prefer immutable memory as default function arguments, more expressive control flow and better ways to define operators.

Current mainstream languages also suffer some semantical problems what we discuss in thesis 2. Implicit conversions usually allowed in this languages but they may lead to unwanted behavior. To avoid such cases in C++ we introduced a wrapper class based solution. The C++11 move semantics may reduce copy operations when implemented properly. We specified an algorithm and implemented a prototype tool to detect possible misuse of the move semantics.

It is well-known that the costs of software development is only the fragment of its all maintenance cycle. In order to reduce maintenance costs, we may demand various services from the programming languages. We analyze such features in thesis 3. As an example, mainstream languages give no support for handle the binary compatibility issues among different library versions. Compile time testing is also rarely supported. We suggest solutions for these problems.

In order to prove that the design decisions we proposed in the earlier theses viable, we implemented Welltype, a prototype programming language that designed according our findings. Our language, is imperative with additional multi-paradigm language elements and available as open source with a full development tool-chain.

References

[1] Á. Baráth and Z. Porkoláb, “Towards safer programming language constructs,” *Studia Universitatis Babes-Bolyai, Informat-*

ica, vol. 60, no. 1, pp. 19–34, 2015. http://www.cs.ubbcluj.ro/~studia-i/contents/2015-1/02-BarathPorkolab.pdf.

[2] Á. Baráth and Z. Porkoláb, "Automatic checking of the usage of the C++ 11 move semantics.," *Acta Cybernetica*, vol. 22, no. 1, pp. 5–20, 2015. https://doi.org/10.14232/actacyb.22.1.2015.2.

[3] Á. Baráth and Z. Porkoláb, "Life without implicit casts: safe type system in C++," in *Proceedings of the 7th Balkan Conference on Informatics Conference*, p. 6, ACM, 2015. https://doi.org/10.1145/2801081.2801114.

[4] Á. Baráth and Z. Porkoláb, "Attribute-based checking of C++ move semantics," in *Proceedings of the 3rd Workshop on Software Quality Analysis, Monitoring, Improvement and Applications (SQAMIA) 2014, Lovran, Croatia, September 19-22, 2014.*, pp. 9–14, 2014. http://ceur-ws.org/Vol-1266/SQAMIA2014_Paper2.pdf.

[5] Á. Baráth and Z. Porkoláb, "Compile-time unit testing," in *Fourth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications SQAMIA 2015*, pp. 1–7, 2015. http://ceur-ws.org/Vol-1375/SQAMIA2015_Paper1.pdf.

[6] Á. Baráth and Z. Porkoláb, "Welltype: Language elements for multiparadigm programming," in *Position Papers of the 2017 Federated Conference on Computer Science and Information Systems*, pp. 91–101, 2017. http://dx.doi.org/10.15439/2017F546.

[7] Á. Baráth and Z. Porkoláb, "Domain-specific languages with custom operators," in *proceedings of The 9th International Conference on Applied Informatics*, 2014.

[8] Á. Baráth and Z. Porkoláb, "Detecting binary incompatible software components using dynamic loader," *Studia Universitatis Babes-Bolyai, Informatica*, vol. 63, no. 1, pp. 51–63, 2018. https://doi.org/10.24193/subbi.2018.1.04.

[9] Á. Baráth, “Welltype project web page.” http://repo.hu/projects/welltype, 2018.

[10] Á. Baráth, “Move semantics checker.” http://baratharon.web.elte.hu/movesem/, 2014.

[11] “Vulnerability summary for cve-2014-1266..” http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266, 2014.

[12] S. S. Laurent, *Introducing Erlang: Getting Started in Functional Programming*. " O’Reilly Media, Inc.", 2017.

[13] B. Stroustrup, *The C++ programming language, 4th Edition*. Addison-Wesley, 2013.

[14] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[15] B. Stroustrup, *Design and Evolution of C++*. Addison-Wesley, 1994.

[16] S. R. Schach, *Object-oriented and classical software engineering*, vol. 6. McGraw-Hill New York, 2002.