Position papers of the Federated Conference on Computer Science and Information Systems, pp. 97–101
DOI: 10.15439/2017F546
ISSN 2300-5963 ACSIS, Vol. 12

Welltype: Language elements for multiparadigm programming

Áron Baráth
Eötvös Loránd University
H-1117 Budapest, Hungary
Email: baratharon@caesar.elte.hu

Zoltán Porkoláb
Eötvös Loránd University
H-1117 Budapest, Hungary
Email: gsd@caesar.elte.hu

***Abstract*—Modern programming languages try to provide a balance between flexibility to support rapid development and implementing as much validation on the program as possible to avoid expensive runtime errors. This trade-off is reflected in the language syntax, the type system and even in the method how the program produces the runtime binary. While the balance seems to be slightly moved today from safety to effectiveness, there is still a high demand for thoroughly checked, safe, but still effective programming languages. In this paper we introduce our experimental, imperative programming language, Welltype, which is designed to demonstrate that effective development can be accommodated with increased safety. Our language design decisions are based on current real-life problems and their solutions. We describe key features such as syntax improvement, fail-safe type system, and binary compatibility via dynamic linking.**

I. Introduction

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 runtime checks are necessary.

In this paper we present the important features of our experimental programming language, Welltype. This language is aimed to prove that clear syntax, strict semantics and strong type system can still provide a friendly language interface for the programmer.

The clear syntax will help the programmers to understand the code after the development. Benefits during maintenance is guaranteed this way in contrast of a language with a more exotic syntax. A tense syntax can easily overwhelm the understanding of the code.

The strict semantics declares that a construction will mean the same regardless of the syntax context. In practice, when a construction indicates multiple but different things, the intention of the code fragment will be ambiguous. Also, it costs some time to a third person to solve the ambiguity. The *strict semantics* appellation is the bridge between the clear syntax and the strong type system: since the types are explicit in the code, and a construction means only one thing, the intention behind a specific code fragment is certain.

The strong type system that Welltype uses is much less permissive that the type system in C or C++. We present the type system in Section IV. The key feature is deeply validated types across dynamic linking.

The Welltype language is designed to be as safe as possible with comfortable language features as described above. We made our design decisions based on current and relevant issues in order to fix them with minimal syntactical and semantical overhead. We present details about the key elements and we make conclusions on each of them.

The paper is organized as follows: In Section II we present the Welltype language, and we give a short overview of the key language features. In each of the following 3 sections we focus on one significant language element as an example. In Section III we concentrate on language syntax. Section IV argues for our design decisions on the type system. In Section V we give a short introduction to the binary compatibility and we show how Welltype handles it. In Section VI we show the related work on language safety. Our paper concludes in Section VII.

II. Overview of Welltype

The Welltype language is an imperative programming language extended with generic and functional programming elements. It is designed to be safe and feature rich in the same time. The syntax is similar to the C++ with improvements. The structure of the source file is redesigned to meet the *safe* requirement. A Welltype source consists of blocks and metadata directives. A block can be *declaration*, *import*, *export*, or *function definition* block. The first three contain declarations, for example functions, operators, records, enums, algebraic data types, exceptions. Note that the imported/exported declarations will be deeply validated (the mechanism behind is discussed in Section V) during the program loading.

The body of a function can contain assignment statements, assertion statements (which can be turned off in release build), `return` statements, `raise` statements (to raise exceptions), conditional statements (`if-elif-else`), loops (`for`, `while`, `do-while`, `foreach`), and switch on algebraic data types and enumeration types (`switch-case`). The complete

grammar is available on the Welltype website [21]. Note that an assignment is not an expression. Furthermore, the semantics disallows free expressions in the code. This improvement (which is clearly a restriction as well) can prevent serious problems on the code snippet that can be seen on Figure 1. The original code was a correct function call, but somehow, the `some_function` identifier is lost from the source code, and the remained statement is still a valid C++ code, but its meaning is totally different. Welltype does not allow the second construction, so the lost `some_function` identifier will cause a compilation error.

```
// original code: function call
some_function(param1, param2);

// errorneous code: sequence operator
(param1, param2);
```

Fig. 1. Error caused by lost function name.

Welltype functions can have exception handler clauses. This syntax decision aimed to keep the source of the function as tidy as possible. Languages like C++ and Java allows to place exception handlers almost anywhere. The problem with this is that the ordinary execution flow will interweave with exception handler fragments that are usually not executed. It can mislead the programmer, and increase the complexity.

In the perspective of the execution, the `raise` statement –and also other sources of exceptions– will find the first exception handler clause on the call stack, and rewind it until the exception handler, and continues. The `raise` statement is very similar to `throw` in C++, but in Welltype programs cannot *throw* exception but they can *raise* them.

For instance, an `IndexOutOfRangeException` will be raised when a `string` or a `seq` is misindexed.

The essential part of the Welltype language is its strong type system. The type system does no support implicit casts, thus the data flow is more followable, and provides faster function lookup. Other restrictions are also encoded into the type system, for example the *mutable* property of a value (variable, or derived value). The Welltype language takes mutable and immutable properties seriously, and in contrast of C++ the *mutable* types should be explicitly tagged – while in C++ the *immutable* (`const`) types should be tagged. We made analysis on some software to determine the ratio of the mutable and immutable function parameters [19]. As an example results on TinyXML [18] can be seen on Figure 2. We concluded that two important reasons support the `mutable`-style instead of the `const`-style. First, much less keywords are necessary; second, if a `mutable` keyword is missing it will cause compilation error at the right place.

An interesting phenomenon can be perceived when we compare the `const`-style and the `mutable`-style approaches. In the `const` world, if the programmer forgets to qualify the function parameter, it will be mutable by default, thus modification may occur. To preserve the parameter (especially object parameters) from modifications, the programmer must explicitly indicate the intention. In the `mutable`, if the programmer forget to quality the function parameter and the function wants to modify the parameter, the compiler will emit an error message due to it is forbidden. To allow modifications of a parameter, the programmer must explicitly indicate it. The Welltype follows the immutable-by-default rule when introducing function parameters. This looks to be a larger effort to write the code, but as we have seen earlier, the number of mutable function parameters is the fragment of the number of constant parameters.

Parameters	236
By value	87
Constant	105
Mutable	44

Fig. 2. Function parameter analysis of TinyXML.

III. Syntax

Syntax is always a main matter of safety. Syntax defines the face of the language, helps the programmer if it is intuitive, and gives a support to detect general programming errors.

In C [15] and C++ [16], the syntax is quite permissive. Companies define coding conventions (e.g. Apple[1], Google[2]) that they know (or they think) as good. Some coding conventions can be assumed as broken, or in other words not good enough. For example, a vulnerability called *goto fail* [20], raised because of a duplicated `goto fail` line as can be seen on Figure 3. This vulnerability could be avoidable if the coding rules were adequate.

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

Fig. 3. The affected lines of the *goto fail* error.

Making the braces mandatory in control flow statements (i.e. `if`) in the language itself is a good alternative. It will guarantee much safer constructions (*goto fail* error cannot happen), and makes the code more readable. The Go language [9] always requires braces in all constructions. The Welltype language follows that design decision, but the position of the braces are different. The detailed specification how to place braces is in the Go language specification [10].

[1] `https://developer.apple.com/library-/ios/documentation/General/Conceptual-/DevPedia-CocoaCore/CodingConventions.html`

[2] `https://google.github.io/styleguide/cppguide.html`

The WordPress[3] team changed their PHP coding standard[4] to always require braces in 2013[5], because they saw its benefit. Furthermore, their JavaScript coding standard[6] also requires braces. Many other relevant discussion can be found about the braces, and where to place them. One thread called *Should curly braces appear on their own line?*[7] from *stackexchange.com* has numerous of interesting comments. Nowadays, it is not difficult to find coding standard that enforce the usage of braces; even larger communities using them for obvious reasons. Our motivation was the fact it is in coding standards indeed, but the compiler will not complain when it is omited: better build into the syntax. Requiring something in the coding standard is one thing, but enforcing them to happen is an other. Programmers will invest less effort if the code started to work.

Another perspective is to remove all braces (and other symbols and keywords that can introduce blocks) from the language, like the popular Python [12] language. Python is whitespace sensitive, and the blocks will be automatically recognized from the indentation. On the other hand, relying only on whitespaces is not necessarily the safest way. We could seperate the elements helping the programmer and the elements helping the compiler. In C, C++, Java, PHP, etc and in Welltype, the indentation is for the programmer to be able to read the code – but the braces are for the compiler, because those will clearly define the block. In a whitespace-only language a misindented statement can cause serious bugs, furthermore, the usually invisible spaces and tabs can also break the ”good” indentation, if they are used mixed. And then, no one will know the original intention.

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

Fig. 4. Erroneous *if* statement (extra semicolon breaks the code).

Still, requiring the braces on the language level can prevent other errors as well, not only the *goto fail*-like errors. The code snippet on Figure 4 is an errorneous code (in C, C++, Java, C#). The extra semicolon after the `if`'s closing parenthesis will close the statement, and the block in the next line is just a regular block. That block has no relation to the `if` statement, but it looks like it has. Since in Welltype a block statement is required as the body of the `if`, the `for`, the `while`, the `do-while` and the `foreach` statement, this kind of error is not possible – that code will not compile. We experienced this and similar errors as a recurrent problem commited by beginner programmers. In many cases the compiler error message was not helping neither.

We presented some aspects of why is a good idea to force block statement as the body of the control flow statements: a little syntactical overhead against the clarity and safety. We conclude that this effort has more benefits than disadvantages.

IV. Semantics and type system

Semantics and the type system is the next bastion of a programming language. Many validations will guarantee that the source fit to the language semantics. Most of these are performed by the type system by checking the types. We consider that the type system is an essentional part of a programming language.

One manifestation is to guarantee *const correctness*. The information that a value is mutable or not can improve the code quality, and helps to avoid illegal modifications (e.g. on a read-only memory area). For instance, C++ supports this, but the Java language has really limited support for this. Furthermore, it is hard to force the programmer to write const-correct code [4]. Furthermore, the lambda expressions (that were introduced in C++11) are immutable by default [5] – but that can be modified using the `mutable` lamba declarator [16].

The `mutable` attribute is inherited by the member recursively as it is expected – as well as the immutable attribute. Like all attributes mutable is part of the type system, therefore it will be stored in the binary. It implies that the mutable attribute will be validated during the dynamic linking process, and this guarantees const-correctness across binaries.

The type system can be permissive itself that can lead to serious problems – we highlight here only one of them. A synthetic version of the problem can be seen on Figure 5. The snippet is a valid code in several languages (Java, C++, and C#) and it is broken in all of them.

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

Fig. 5. Infinite loop caused by implicit cast.

This construction is a really nasty one, and the programmer may get a warning message for that. Problem is with the loop condition: it compares a `char` and an `int`. Many languages will promote (or cast) the `char` to `int`, because if they do, the comparison makes sense. But the domain of the two types are different, and the left-hand side will never has a value of `70000`. Therefore, the loop is an infinite loop. In many languages we just cannot avoid this kind of error. However, in C++ we developed a working solution for this problem [11]. It is a subsequent milestone of our endeavor to make languages safer [19]. Welltype solves this problem only with its strong type system: the comparison `ch<70000` is illegal, due to `char` and `int` cannot be compared.

The Welltype language guarantees that all literals have one, and exactly one type. For instance, the type of the the literal

[3] https://wordpress.org/

[4] https://make.wordpress.org/core/handbook/best-practices/coding-standards/php/

[5] https://make.wordpress.org/core/2013/11/13-/proposed-coding-standards-change-always-require-braces/

[6] https://make.wordpress.org/core/handbook/best-practices/coding-standards/javascript/

[7] http://programmers.stackexchange.com/questions-/2715/should-curly-braces-appear-on-their-own-line

`1u` will be `uint`. In C and C++ the literals can have different type depending on the length of the literal. As can be seen in Figure 6 the type of the literal can be different regardless of the suffix. Thus, `sizeof(3000000000)` is not equal to `sizeof(3000000000u)`. In Welltype, the literals that overflow from the domain of its type will cause compile error.

Signed	Type	Unsigned	Type
1	int	1u	unsigned int
2000000000	int	2000000000u	unsigned int
3000000000	long	3000000000u	unsigned int
30000000000	long	30000000000u	unsigned long

Fig. 6. Type of literals in C++ (compiled with g++ LP64 on 64-bit system).

As mentioned earlier, literals in Welltype have exactly one type. The type can be determined in a deteministic way using only the source code itself without its context. That is, if we look at only a single literal, we know its type certainly. For instance, the literal `1234` has the type `int`; the literal `250ub` has the type `ubyte`. A quick overview can be seen in Figure 7. Note that in Welltype a number literal may have underscore (_) in it, and it will not change the value of the literal. Underscores will be removed automatically from the literals before further processing. However, the literal `270ub` – in contrast of C/C++ – is illegal, and causes a compilation error. Take the `5000000000u` literal in C; this suggests that it is an `unsigned int` after the suffix, but the literal overflows the `unsigned int` types, and became an `unsigned long long`.

Literal	Type
10	int
20_l	long
30_u	uint
40_ul	ulong
50_ib	byte
60_ub	ubyte
70_h	short
80_uh	ushort

Fig. 7. Quick overview of integer literal suffixes.

Since Welltype forbids implicit conversion, problems like on Figure 5 are not possible. We conclude that handling type correctly will decrease the errorneousness of the code. Additionally, we earlier presented a technique for C++ to avoid implicit conversions [11].

V. Linking (binary compatibility)

We talk about binary compatibility when we have multiple releases of a software module. Suppose that we can successfully compile all versions. The question is whether the client binary that uses one of the compiled versions is able to use the other binary versions without modification? When the answer is *yes*, we can say the versions are binary compatible. When we compile and link multiple modules to – for example – an executable we can fix the problem easily: just recompile the affected modules. However, when we have no control on compilation and linking, like in case of C and C++ dynamic libraries, it is not trivial to decide which versions of the library are compatible. This is a common situation when library maintainers should modify a library and produce a shared object which will be used on-demand by unknown users. Breaking binary compatibility in this case causes the client program to crash. However, the problem is not limited to C or C++, this is a real-life issue, for example, in Java as well [2], [6].

Binary incompatibility can happen in more mystical reasons to. In one of our projects (it was the Welltype compiler itself) there was multiple C++ sources that used the `FlexLexer.h`, which is a system-wide header belongs to the *flex* tool. The source files have been already compiled, when a system upgrade was performed. The `flex` package was upgraded as well, and the `FlexLexer.h` was changed (two fields lost the reference qualifier). After that, one of the source files that use the `FlexLexer.h` changed, and a build was performed. Since the system-wide headers are not a real dependencies in the build system, only the changed C++ files were recompiled – then the executable was linked. Thus the executable crashed, because the changes of the `FlexLexer.h` was not applied in all source, and the old object files became incompatible with the new ones. The problem originated to the linker, because it cannot recognized such inconsistencies.

The Welltype language aimed to avoid binary incompatibilities: since the Welltype dynamic loader **deeply 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 deeply validated, which consists of name of the record, and number, type and name of the fields. The reason why the Welltype validates so deep, is to detect the changes. For instance, if two fields in a record are swapped, the client program will still contain code for the original record, but the other side assumes the modified version of the record – thus, the program will not work. Therefore, it is reasonable to detect this kind of changes at load time, and the runtime environment can refuse to load the incompatible program.

This mechanism ensures that binary incompatibilities caused by a side effect of the language will not occur, since the binary interface is well-defined. For example, in C++ if the programmer uses only the public API of a class, the compiler may generate inline code that will percolate code from the library into the client program. This is an easy way to make the client program binary incompatible to the next version of the library. However, Welltype prevents this situation, and the binary interface cannot be bridged like in C++.

VI. Related work

Our research included improving and extending existing mainstream languages. We extended the type system in C++ [11], to forbid implicit casts.

We researched how to check the correct usage of the move semantics in C++11 [1]: the move operation is introduced to improve efficiency, but unwanted copy operations may hidden inside. This area related to the detect heavy runtime overheads at compilation time. We developed a prototype tool (based on Clang Tooling [17], [8]), that can identify when the copy-semantics is used instead of move-semantics, and it is not reasonable. Therefore, the unnecessary copy operations can be eliminated from the code.

We developed an other C++ extension to do compile-time unit testing [7]. This work was inspired by the motto *do as many work at compile-time as possible*. Furthermore, when the unit tests are performed at compilation time, the code is guaranteed as tested, so it is not "too bad"; also, after changed the code will compile again, if it passes the tests, and it cannot be omitted.

The idea to introduce a much restricted type system is not rare. The Scala language also uses a more complex type system with immutable types to increase security [14]. The Rust language represent an other approach to increase security by being resource-safe [13]. Also, this attitude evolving in C++ by the *C++ Core Guidelines* [3]. When a language supports more safety feature, the costs of static analysis will be lower. For example, the C++ langauge is permissive enough to grow static analysis into a very complex task (CppCheck, Lint, Clang Static Analyzer) – static analysing tools aimed to improve code quality. It would be great if the compiler could perform such task during the compilation.

VII. Conclusion

In this paper we presented our experimental programming language, Welltype. As are earlier researches discovered many critical issues in modern programming languages supposed to be safe, we decided to create a prototype language to show that there is a feasible trade-off between safety and the ease of use. We presented revealing examples on three major parts of Welltype: rigorous syntax, strict type system, advanced linking features.

Welltype provides strict syntactical rules to minimize errors caused by typos. Missing or duplicated semicolons, braces or identifiers cause syntax error. The strict syntax also helps code comprehension.

Welltype semantics supposes all function parameters immutable. Measurements on projects implemented in mainstream languages prove that function parameters are mostly supposed not to modify, but mainstream languages do not support this. They threat parameters mutable by default, and may can be changed to immutable only by additional effort. Implicit conversions are other source of runtime errors. Such situations are hard to avoid and even harder to investigate. Welltype's strict type system avoids this kind of problems. By this approach increases safety it has also a positive effect on compilation time as implicit conversions are significant source increased compile time.

Binary compatibility is an issue poorly recognized by language designers, but can cause serious headache for maintainers of large software projects. When already compiled clients are linked against different versions of libraries, incompatible library versions can cause the client code to crash or even worse, to running in undefined way. This problem frequenty occurs with C/C++ programs using dynamic libraries, but the issue is not limited to C++, also happens in Java and other languages. Welltype deeply validates modules to link and forbids incompatible usage.

The Welltype compiler is freely available and testable [21]. The language serves as a working prototype to show how safety in various dimensions can extend modern programming languages.

References

[1] Baráth, Á., Porkoláb, Z.: *Automatic Checking of the Usage of the C++ 11 Move Semantics*. ACTA CYBERNETICA-SZEGED 22: pp. 5–20. (2015)

[2] Dietrich, J., Jezek, K., Brada, P.: *Broken promises: An empirical study into evolution problems in java programs caused by library upgrades*. Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on. IEEE, (2014), https://doi.org/10.1109/CSMR-WCRE.2014.6747226

[3] Stroustrup, B.: *C++ Core Guidelines* `https://github.com/isocpp/CppCoreGuidelines`

[4] Cline, M. P., Lomow, G. and Girou, M.: *C++ FAQs*. Pearson Education (1998)

[5] Järvi, J., Freeman, J.: *C++ lambda expressions and closures*. Science of Computer Programming 75.9 (2010): 762-772. https://doi.org/10.1016/j.scico.2009.04.003

[6] Savga, I., Rudolf M., Goetz, S.: *Comeback!: a refactoring-based tool for binary-compatible framework upgrade*. Companion of the 30th international conference on Software engineering. ACM, (2008), https://doi.org/10.1145/1370175.1370198

[7] Baráth, Á., Porkoláb, Z.: *Compile-time Unit Testing*. 4th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications pp. 1–7. ISBN 978-961-248-485-9 (2015)

[8] Duffy, Edward B., Brian A. Malloy, and Stephen Schaub. *Exploiting the Clang AST for analysis of C++ applications.* Proceedings of the 52nd Annual ACM Southeast Conference. 2014.

[9] Alan A. A. Donovan, Brian W. Kernighan. *The Go Programming Language.* Addison-Wesley Professional, ISBN: 978-0134190440 (2015)

[10] Go Programming Language Specification. `https://golang.org/ref/spec`

[11] Baráth, Á., Porkoláb, Z.: *Life without implicit casts: safe type system in C++*. Proceedings of the 7th Balkan Conference on Informatics, ISBN 978-1-4503-3335-1 (2015), https://doi.org/10.1145/2801081.2801114

[12] Summerfield, M.: *Programming in Python 3: a complete introduction to the Python language*. Addison-Wesley Professional, ISBN 978-0321680563 (2010)

[13] Matsakis, Nicholas D., and Felix S. Klock II.: *The rust language.* ACM SIGAda Ada Letters. Vol. 34. No. 3. ACM, (2014) http://doi.org/10.1145/2692956.2663188

[14] Layka, V., and Pollak, D.: *Scala Type System.* In Beginning Scala (pp. 133-151). Apress. (2015)

[15] Kernighan, B. W., and Ritche, D. M.: *The C programming language*. Vol. 2. Englewood Cliffs: prentice-Hall (1988)

[16] Stroustrup, B. *The C++ Programming Language, 4th Edition*. Addison-Wesley (2013)

[17] Klimek, M.: *The Clang AST – a Tutorial*. `http://llvm.org-/devmtg/2013-04/klimek-slides.pdf` (2013)

[18] TinyXML. `http://www.grinninglizard.com/tinyxml2`

[19] Baráth, Á., Porkoláb, Z.: *Towards Safer Programming Language Constructs*. Studia Univ. Babes-Bolyai Ser. Inf. LX:(1) 19-34 (2015)

[20] Vulnerability Summary for CVE-2014-1266. http://web.nvd.nist.gov-/view/vuln/detail?vulnId=CVE-2014-1266.

[21] Welltype web page. `http://baratharon.web.elte.hu/-welltype/`